

ARCoSS

LNCS 16506

Sebastian Junges
Guy Katz (Eds.)

Tools and Algorithms for the Construction and Analysis of Systems

32nd International Conference, TACAS 2026
Held as Part of the International Joint Conferences
on Theory and Practice of Software, ETAPS 2026
Turin, Italy, April 11–16, 2026, Proceedings, Part II

2 Part II



 Springer

OPEN ACCESS

Lecture Notes in Computer Science

16506

Founding Editors

Gerhard Goos
Juris Hartmanis

Editorial Board Members

Elisa Bertino , USA
Wen Gao, China

Bernhard Steffen , Germany
Moti Yung , USA

Advanced Research in Computing and Software Science

Subline of Lecture Notes in Computer Science

Subline Series Editors

Giorgio Ausiello, *University of Rome 'La Sapienza', Italy*
Vladimiro Sassone, *University of Southampton, UK*

Subline Advisory Board

Susanne Albers, *TU Munich, Germany*
Benjamin C. Pierce, *University of Pennsylvania, USA*
Bernhard Steffen , *University of Dortmund, Germany*
Deng Xiaotie, *Peking University, Beijing, China*
Jeannette M. Wing, *Microsoft Research, Redmond, WA, USA*

More information about this series at <https://link.springer.com/bookseries/558>

Sebastian Junges · Guy Katz
Editors

Tools and Algorithms for the Construction and Analysis of Systems

32nd International Conference, TACAS 2026
Held as Part of the International Joint Conferences
on Theory and Practice of Software, ETAPS 2026
Turin, Italy, April 11–16, 2026
Proceedings, Part II

Editors

Sebastian Junges 
Radboud University
Nijmegen, The Netherlands

Guy Katz 
The Hebrew University of Jerusalem
Jerusalem, Israel



ISSN 0302-9743

ISSN 1611-3349 (electronic)

Lecture Notes in Computer Science

ISBN 978-3-032-22748-5

ISBN 978-3-032-22749-2 (eBook)

<https://doi.org/10.1007/978-3-032-22749-2>

© The Editor(s) (if applicable) and The Author(s) 2026. This book is an open access publication.

Open Access This book is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this book or parts of it.

The images or other third party material in this book are included in the book's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the book's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

This work is subject to copyright. All commercial rights are reserved by the author(s), whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Regarding these commercial rights a non-exclusive license has been granted to the publisher.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Please refer to the chapters to see the exact Creative Commons Attribution licenses that apply in each case.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

If disposing of this product, please recycle the paper.

ETAPS Foreword

Welcome to the 29th edition of ETAPS, which took place as an on-site event in Turin, Italy during April 11–16, 2026!

ETAPS 2026 was the 29th instance of the International Joint Conferences on Theory and Practice of Software (ETAPS). ETAPS is an annual federated conference established in 1998, and consists of four main conferences: ESOP, FASE, FoSSaCS, and TACAS. Each conference has its own Program Committee (PC) and its own Steering Committee (SC). The ETAPS main conferences cover various aspects of software systems, ranging from theoretical computer science to foundations of programming languages, tools and algorithms for system analysis, and formal approaches to software engineering. Organizing these conferences in a coherent, highly synchronized conference programme enables researchers to participate in an exciting event, having the possibility to meet many colleagues working in different directions in the field, and to easily attend talks of different conferences. In addition to its four main conferences, ETAPS 2026 also hosted fifteen satellite workshops and two colocated events, which together further attracted many researchers from all over the globe.

ETAPS 2026 received 456 submissions in total, 138 of which were accepted, yielding an overall acceptance rate of 30%. Out of the 138 accepted papers, 16 papers were selected as ETAPS distinguished papers. I thank all the authors of submitted papers for their interest in ETAPS, all the reviewers for their reviewing efforts, the PC members for their contributions, and in particular the PC (co-)chairs for their hard work in running this entire intensive process in a constructive, objective and timely manner. I congratulate all authors of the ETAPS 2026 accepted papers!

ETAPS 2026 featured the unifying invited keynotes by

- Monika Henzinger (Institute of Science and Technology Austria, Austria), delivering a talk about “Guarding Privacy Over Time: Challenges and Solutions in Continuous Data Observation”,
- Einar Broch Johnsen (University of Oslo, Norway), discussing “Formal Methods Meet Digital Twins: Challenges and Opportunities”.

ETAPS 2026 hosted the invited keynote speakers

- Christel Baier (Technische Universität Dresden, Germany) for FoSSaCS, presenting “Verification of Infinite-horizon Properties of Dynamic Bayesian Networks”,
- Guy Van den Broeck (University of California, Los Angeles, USA) for TACAS, introducing “Symbolic Reasoning in the Age of Large Language Models”.

The ETAPS 2026 invited tutorials were provided by

- Mieke Massink (CNR-ISTI Pisa, Italy) on “Model Checking in Space with Applications to Medical Image Analysis”,
- Leonardo de Moura (Amazon Web Services, USA) surveying “The Lean Programming Language and Theorem Prover”.

The ETAPS 2026 programme also featured a lively Ask-Me-Anything session, interactive tool demos, a Diversity, Equity, and Inclusion session, SV-Comp and Test-Comp community building events, and the ETAPS industry day. The goal of the ETAPS industry day is to bring industrial practitioners into the heart of the research community and to catalyze the interaction between industry and academia. The ETAPS 2026 industry day was organized by Giorgio Audrito (University of Turin, Italy), Sean Kauffman (Queen's University, Kingston, Canada), and Nikolai Kosmatov (Thales Research and Technology, Palaiseau, France).

ETAPS 2026 was organized by the Department of Computer Science of the University of Turin, which is the center for coordinating research, teaching, dissemination and technological transfer in computer science in Turin, Italy. The department covers both methodological and application oriented aspects of computer science, and performs research in several interdisciplinary areas. This is reflected in the collaborations with other research centers and companies in many scientific areas and in its participation in national, European and international projects.

ETAPS 2026 was further supported by the following associations and societies: ETAPS e. V. (the ETAPS Association), EATCS (European Association for Theoretical Computer Science), EAPLS (European Association for Programming Languages and Systems), and EASST (European Association of Software Science and Technology).

The ETAPS Steering Committee consists of an Executive Board, and representatives of the individual ETAPS conferences, as well as representatives of EATCS, EAPLS, and EASST. The Executive Board consists of Laura Kovács (TU Wien, chair), Andrzej Wąsowski (IT University of Copenhagen, vice-chair), Thomas Noll (RWTH Aachen, treasurer), Arnd Hartmanns (University of Twente, artifact evaluation coordinator), Barbara König (University of Duisburg-Essen, proceedings coordination), Caterina Urban (Inria, PhD activities), Elizabeth Polgreen (University of Edinburgh, social media), Jan Kofron (Charles University Prague, organisational support, website), Jan Křetínský (Masaryk University Brno and TU Munich, diversity & inclusion), and Marieke Huisman (University of Twente, blog, awards). Further members of the ETAPS Steering Committee committee are: Robbert Krebbers (Radboud University Nijmegen), Azalea Raad (Imperial College London), Luís Caires (Tecnico ULisboa), Elvira Albert (Universidad Complutense de Madrid), Corina Păsăreanu (Carnegie Mellon University), Erika Ábrahám (RWTH Aachen), Marsha Chechik (University of Toronto), Marie-Christine Jakobs (LMU Munich), Nathalie Bertrand (Inria Rennes), Stefan Milius (Friedrich-Alexander Universität Erlangen-Nürnberg), Alexandra Silva (Cornell University), Joël Ouaknine (MPI-SWS Saarbrücken), Andrzej Murawski (University of Oxford), Sebastian Junges (Radboud University Nijmegen), Guy Katz (The Hebrew University of Jerusalem), Christian Schilling (Aalborg University), Naijun Zhan (Peking University), Joost-Pieter Katoen (RWTH Aachen and University of Twente), Dirk Beyer (LMU Munich), Fabrice Kordon (Sorbonne University Paris), Laure Petrucci (Université Paris 13), Peter Y.A. Ryan (University of Luxembourg), Claudio Menghi (University of Bergamo and McMaster University Hamilton), Mark Lawford (McMaster University Hamilton), Maurice ter Beek (CNR-ISTI Pisa), Ferruccio Damiani (University of Turin), Kim Guldstrand Larsen (Aalborg University), Bernhard Beckert (KIT Karlsruhe), Mattias Ulbrich (KIT Karlsruhe), Reiko Heckel (University of Leicester), Vladimiro Sassone

(University of Southampton), Anton Wijs (Eindhoven University of Technology), and Nikolai Kosmatov (Thales Research and Technology, Palaiseau).

The ETAPS 2026 local organization team consisted of Maurice ter Beek (CNR-ISTI Pisa, general co-chair), Ferruccio Damiani (University of Turin, general co-chair), Barbara Boni (Synesthesia Turin, local organization chair), Vincenzo Ciancia (CNR-ISTI Pisa, satellite events co-chair), Luca Paolini (University of Turin, satellite events co-chair), Maria Tacconi (Synesthesia Turin, satellite events co-chair and publicity co-chair), Francesco Brocero (Synesthesia Turin, web co-chair and volunteers co-chair), José Proença (University of Porto, web co-chair), Gianluca Torta (University of Turin, publicity co-chair and local proceedings co-chair), Lucy James (Synesthesia Turin, sponsor chair), Giovanna Broccia (CNR-ISTI Pisa, local proceedings co-chair), Giorgio Audrito (University of Turin, volunteers co-chair), Riccardo Sieve (UiO Oslo, volunteers co-chair), and Reiner Hähnle (TU Darmstadt, wine chair).

I would like to take this opportunity to thank all authors, keynote speakers, invited tutorial speakers, and attendees. Special thanks goes to the organizers of the ETAPS 2026 satellite workshops and colocated events. ETAPS 2026 is grateful for the generous support of Amazon Web Services, AccessiWay, Camera di Commercio Industria Artigianato e Agricoltura di Torino, the Department of Computer Science of the University of Turin, Springer Nature, and Turismo Torino e provincia Convention Bureau. I thank our general co-chairs Maurice ter Beek (CNR-ISTI Pisa) and Ferruccio Damiani (University of Turin), who made it all happen in Turin, and their local organization team for their enormous efforts to make ETAPS 2026 a fantastic event. I am especially grateful to Barbara Boni, Maria Tacconi, and Lucy James (Synesthesia Turin) for handling the organizational process in a smooth and reliable way. Last but not least, a big thanks to Jan Kofroň for all his help as an ETAPS Fellow and providing online presence support for the ETAPS conferences and the ETAPS Association.

I hope you all enjoyed ETAPS 2026!

April 2026

Laura Kovács
ETAPS SC Chair, President of the ETAPS
Association

Preface

This two-volume proceedings contains the papers presented at the 32nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2026). TACAS 2026 was part of the 29th International Joint Conferences on Theory and Practice of Software (ETAPS 2026), which was held between April 11–16, 2026, in Turin, Italy.

TACAS is a forum for researchers, developers and users interested in rigorous tools and algorithms for the construction and analysis of systems. The conference aims to bridge the gaps between different communities with this common interest and to support them in their quest to improve the utility, reliability, flexibility, and efficiency of tools and algorithms for building systems. TACAS 2026 interleaves and integrates various disciplines, including formal verification of software and hardware systems, static analysis, probabilistic programming, program synthesis, concurrency, testing, simulations, verification of machine learning/autonomous systems, cyber-physical systems, SAT/SMT solving, automated and interactive theorem proving, and proof checking.

There were four submission categories for TACAS 2026:

1. Regular research papers, identifying and justifying a principled advance to the theoretical foundations for the construction and analysis of systems.
2. Case study papers, describing the application of techniques developed by the community to a single problem or a set of problems of practical importance, preferably in a real-world setting.
3. Regular tool papers, presenting a novel tool or a new version of an existing tool, built using novel algorithmic and engineering techniques.
4. Tool demonstration papers, demonstrating a new tool or application of an existing tool on a significant case-study.

Regular research, case study, and regular tool paper submissions were restricted to 16 pages, whereas tool demonstration papers were restricted to 6 pages, excluding the bibliography and the data availability statement. An extra page was allowed for the final version of each accepted paper.

TACAS 2026 received 190 submissions, consisting of 117 regular research papers, 11 case study papers, 33 regular tool papers, and 16 tool demonstration papers. 13 papers were desk-rejected for being out-of-scope or for violating anonymity requirements as outlined in the CfP. Each submission was assigned for review to at least three Program Committee (PC) members, who made use of subreviewers. In total, we obtained 566 reviews, of which 260 were contributed by subreviewers and/or external experts. Regular research papers were reviewed in double-blind mode, whereas case study, regular tool, and tool-demonstration papers were reviewed using a single-blind reviewing process.

Similarly to previous years, it was possible to submit an artifact alongside a paper. Artifact submission was mandatory for regular tool and tool demo papers, and was voluntary, but encouraged, for regular research and case study papers. An artifact might

consist of a tool, models, proofs, or other data required for validation of the results of the paper. The Artifact Evaluation Committee (AEC) was tasked with reviewing the artifacts, based on their documentation, ease of use, and, most importantly, whether the results presented in the corresponding paper could be accurately reproduced. When relevant, AEC reviewers also assessed whether the tool could easily be extended to novel examples and future research. The evaluation was carried out using either a standardized virtual machine or a Docker image to ensure the consistency of the results. The virtual machine used this year included, for the first time, support for ARM architectures, in addition to x86 architectures supported previously. Exceptions were made for those artifacts that had special hardware or software requirements.

Artifact evaluation at TACAS 2026 consisted of two rounds. The first round implemented the mandatory artifact evaluation of regular tool and tool demonstration papers; this round was carried out in parallel with the work of the PC. The judgment of the AEC was communicated to the PC and weighed in their discussion. In particular, regular tool papers and tool demo papers whose artifacts did not earn the “Available” and “Functional” badges were rejected. The second round of artifact evaluation carried out the voluntary artifact evaluation of regular research and case study papers, and took place after paper acceptance notifications were sent out; authors of accepted regular research and case study papers were invited to submit artifacts for this purpose. In both rounds, the AEC provided 2–4 reviews (typically 3) and a summary meta-review for each artifact, and communicated anonymously with the authors to resolve any apparent technical issues. In total, the AEC performed 227 reviews and evaluated a total of 77 artifacts regarding their availability, functionality, and/or reusability. 48 artifacts were submitted to the first round and 29 to the second round. Papers with an artifact that were successfully evaluated include one or more badges on the first page, certifying the respective properties. 71 artifacts were awarded the “Available” badge, 66 the “Functional” badge and 34 were awarded the “Reusable” badge.

All authors were given a chance to provide a rebuttal response to questions raised in the PC discussions. Using the review reports and rebuttals, the PC had a thorough discussion on each paper. For regular tool and tool demonstration papers, the PC also discussed the corresponding artifact, using the AEC recommendations. The papers were discussed during a short but lively online discussion phase with over 1,400 posts, followed by a virtual meeting with the area chairs and PC chairs. As a result, the PC decided to accept 56 papers, which consisted of 34 regular research papers, 15 regular tool papers, 3 case study papers, and 4 tool demonstration papers. This corresponds to an overall acceptance rate slightly lower than 30%. Each accepted paper at TACAS 2026 had either all positive reviews and/or a “championing” PC member who argued in favor of accepting the paper. All papers received a meta-review with a summary of the discussion phase.

TACAS 2026 also hosted SV-COMP 2026, the 15th International Competition on Software Verification. This event evaluated 72 tools for automatic verification of C and Java programs and 19 tools for witness validation, where 43 verification and 13 validation tools were registered and actively supported by development teams. Besides the languages C and Java, the competition now includes a category for the language SV-LIB, a new exchange format for verification of imperative programs. This demonstration category attracted 3 verifiers and 1 validator. In total, SV-COMP included 12 verifiers and

3 validators that participated for the first time. The TACAS 2026 proceedings contain a competition report by the SV-COMP chairs and 15 short papers selected by the competition jury. The 15 short papers were reviewed by a separate Program Committee (jury); each was assessed by at least three jury members. Two sessions in the TACAS 2026 program were reserved for SV-COMP: (1) a presentation session on Monday afternoon with a report by the competition chairs and summaries by the development teams of participating tools, and (2) an open community meeting on Thursday afternoon.

We would like to thank everyone who helped to make TACAS 2026 a success. We thank the authors for submitting their papers to TACAS 2026. The PC members and additional reviewers did an excellent job in reviewing papers: they provided detailed reports and engaged in the PC discussions. We thank our six area chairs: Roderick Bloem, Dana Drachler Cohen, Anastasia Mavridou, Dave Parker, Natasha Sharygina and Yoni Zohar, who proved invaluable in moderating the various PC discussions and pushing them forward. Likewise, we thank the AEC members for their thorough evaluation of all artifacts. We thank the TACAS Steering Committee, and especially its chair, Joost-Pieter Katoen, for his valuable advice. We are grateful to the ETAPS Steering Committee, and in particular its chairs, Marieke Huisman and Laura Kovács, for supporting our changes and suggestions on the TACAS 2026 review process and final program; to Jan Kofroň, for help in updating and maintaining the web presence; to Arnd Hartmanns as AE coordinator, and to Barbara König as publication coordinator. We also acknowledge the support provided by the EasyChair and HotCRP developers, and by Loes Kruger in checking the camera-ready versions. Lastly, we would like to thank the overall organization team of ETAPS 2026, in particular the proceedings co-chairs Giovanna Broccia and Gianluca Torta, for their help in creating these proceedings. Some of the Alt Texts used in the figures in this volume were generated using the publisher's automation, they were not author-generated.

April 2026

Sebastian Junges
 Guy Katz
 PC Chairs
 Guy Amir
 Matthias Volk
 AEC Chairs
 Dirk Beyer
 Jan Strejček
 SV-COMP Chairs

Organization

Program Committee Chairs

Sebastian Junges

Guy Katz

Radboud University, Netherlands

The Hebrew University of Jerusalem, Israel

Area Chairs

Roderick Bloem

Dana Drachler Cohen

Anastasia Mavridou

Dave Parker

Natasha Sharygina

Yoni Zohar

Graz University of Technology, Austria

Technion, Israel

KBR/NASA Ames Research Center, USA

University of Oxford, UK

Università della Svizzera italiana, Italy

Bar-Ilan University, Israel

Program Committee

Erika Ábrahám

S. Akshay

Elvio Gilberto Amparore

Étienne André

Christel Baier

Haniel Barbosa

Clark Barrett

Dirk Beyer

Armin Biere

Luca Bortolussi

Milan Česka

Supratik Chakraborty

Mingshuai Chen

Yu-Fang Chen

Wei-Ngan Chin

Alessandro Cimatti

Rayna Dimitrova

RWTH Aachen University, Germany

IIT Bombay, India

Università di Torino, Italy

Nantes Université, France

TU Dresden, Germany

Universidade Federal de Minas Gerais, Brazil

Stanford University, USA

LMU Munich, Germany

University of Freiburg, Germany

University of Trieste, Italy

Brno University of Technology, Czechia

IIT Bombay, India

Zhejiang University, China

Academia Sinica, Taiwan

National University of Singapore, Singapore

Fondazione Bruno Kessler, Italy

CISPA Helmholtz Center for Information
Security, Germany

Dana Fisman

Hadar Frenkel

Ben-Gurion University, Israel

Bar-Ilan University, Israel

Florian Frohn	RWTH Aachen University, Germany
Mirco Giacobbe	University of Birmingham, UK
Arie Gurfinkel	University of Waterloo, Canada
Arnd Hartmanns	University of Twente, Netherlands
Marijn Heule	Carnegie Mellon University, USA
Falk Howar	TU Dortmund University, Germany
Omri Isac	The Hebrew University of Jerusalem, Israel
Nils Jansen	Ruhr University Bochum, Germany
Daniela Kaufmann	TU Wien, Austria
Orna Kupferman	The Hebrew University of Jerusalem, Israel
Alfons Laarman	Leiden University, Netherlands
Aina Niemetz	Stanford University, USA
Petr Novotný	Masaryk University, Czechia
Guillermo Perez	University of Antwerp, Belgium
Andrea Pferscher	University of Oslo, Norway
Kristin Yvonne Rozier	Iowa State University, USA
Ocan Sankur	Mitsubishi Electric R&D Centre Europe, France
Christian Schilling	Aalborg University, Denmark
Anne-Kathrin Schmuck	Max-Planck-Institute for Software Systems, Germany
Mihaela Sighireanu	ENS Paris-Saclay, LMF, France
Sadegh Soudjani	Newcastle University, UK
Jiri Srba	Aalborg University, Denmark
Armando Tacchella	Università di Genova, Italy
Hazem Torfah	Chalmers University of Technology, Sweden
Yakir Vizel	Technion, Israel
Bow-Yaw Wang	Academia Sinica, Taiwan
Naijun Zhan	Institute of Software, Chinese Academy of Sciences, China
Lijun Zhang	Institute of Software, Chinese Academy of Sciences, China
Djordje Zikelic	Singapore Management University, Singapore

Artifact Evaluation Committee Chairs

Guy Amir	Cornell University, USA
Matthias Volk	Eindhoven University of Technology, Netherlands

Artifact Evaluation Committee

Abha Chaudhary	Binghamton University, USA
Abhishek Singh	IIIT Hyderabad, India
Aditya Senthilnathan	Cornell University, USA
Alberto Bombardelli	Fondazione Bruno Kessler, Italy
Alejandro Hernández-Cerezo	Complutense University of Madrid, Spain
Alexander Stekelenburg	University of Twente, Netherlands
Andy Oertel	Lund University, Sweden
Aron Ricardo Perez-Lopez	Stanford University, USA
Aosen Xiong	University of Waterloo, Canada
Arshia Rafieioskouei	Michigan State University, USA
Åsmund Aqissiaq	University of Oslo, Norway
Arild Kløvstad	
Ayaka Yorihiro	Cornell University, USA
Benedikt Peterseim	Universiteit Twente, Netherlands
Benjamin Przybocki	Carnegie Mellon University, USA
Bob Rubbens	University of Twente, Netherlands
Bruno Andreotti	Universidade Federal de Minas Gerais, Brazil
Charles de Haro	École Normale Supérieure, Université PSL, France
Chris Johannsen	Iowa State University, USA
Christoph Wernhard	University of Potsdam, Germany
Clara Rodríguez Núñez	Universidad Complutense de Madrid, Spain
Dapeng Zhi	East China Normal University, Jiangsu University of Technology, China
Divyesh Unadkat	Synopsys Inc, India
Edward Wang	MIT, USA
Elizaveta Pertseva	Stanford University, USA
Étienne André	Nantes Université, France
Evgenii Vinarskii	Télécom SudParis/Institut Polytechnique de Paris, France
Filip Cano	Institute of Science and Technology, Austria
Filip Macák	Brno University of Technology, Czechia
Geoff Sutcliffe	University of Miami, USA
Hanna Lachnitt	Stanford University, USA
Hassan Mousavi	University of Tehran and Tehran Institute for Advanced Studies, Iran
Hichem Rami Ait-El-Hara	OCamlPro, France
Ian Dardik	Carnegie Mellon University, USA
Idan Refaeli	The Hebrew University of Jerusalem, Israel

Jiong Yang	Georgia Institute of Technology, USA
Jonathan Spiegelman	The Hebrew University of Jerusalem, Israel
Juliane Päßler	University of Oslo, Norway
Kartik Sabharwal	University of Iowa, USA
Karthik Nukala	SRI International, USA
Kevin Van de Glind	Eindhoven University of Technology, Netherlands
Leyi Cui	Carnegie Mellon University, USA
Luca Marzari	University of Verona, Italy
Mara Miulescu	Eindhoven University of Technology, Netherlands
Marco Campion	Sorbonne Université, France
Marco Lewis	INRIA, France
Mariem Hammami	UPEC, France
Mark Peyrer	Johannes Kepler University, Austria
Martin Kristjansen	Aalborg University, Denmark
Matthias Hetzenberger	TU Wien, Austria
Max Fan	Cornell University, USA
Mahboubeh Samadi	Tehran Institute for Advanced Studies, Iran
Maria Belen Rodriguez	University of Twente, Netherlands
Maurice Laveaux	Eindhoven University of Technology, Netherlands
Miguel Isabel	Universidad Complutense de Madrid, Spain
Ming kai Miao	The Hong Kong University of Science and Technology (Guangzhou), China
Mohammad Afzal	TCS Research Pune and IIT Bombay, India
Neea Rusch	Uppsala University, Sweden
Nicolas Amat	ONERA — The French Aerospace Lab, France
Nils Lommen	RWTH Aachen, Germany
Oyendrila Dobe	Amazon Web Services, USA
Pablo Gordillo	Complutense University of Madrid, Spain
Philipp Schlehuber-Caissier	Telecom SudParis/Samovar, France
Raya Elsaleh	The Hebrew University of Jerusalem, Israel
Reza Soltani	University of Twente, Netherlands
Robert Modderman	University of Twente, Netherlands
Rui Ge	University of British Columbia, Canada
Samuel Akinwande	Stanford University, USA
Sarat Chandra Varanasi	GE Aerospace Research, USA
Sergei Novozhilov	The Hong Kong University of Science and Technology, China
Simon Robillard	LIRMM, Université de Montpellier & CNRS, France
Syyeda Zainab Fatmi	University of Oxford, UK
Tobias Seufert	University of Freiburg, Germany
Tom Yaacov	Ben-Gurion University, Israel

Tudor Braicu	Cornell University, USA
Vaibhav Mehta	Cornell University, USA
Vlad-Constantin Craciun	Bitdefender, Romania
Weiyi Chen	Purdue University, USA
Wietze Koops	Lund University and University of Copenhagen, Denmark
Xuyang Li	Purdue University, USA
Yonghui Liu	Australian National University, Australia
Yusen Su	University of Waterloo, Canada
Yun Chen Tsai	National Institute of Informatics, Japan
Zvika Berger	Bar-Ilan University, Israel

SV-COMP Organization Committee

Chairs

Dirk Beyer	LMU Munich, Germany
Jan Strejček	Masaryk University, Brno, Czechia

Benchmark Quality Assurance

Zsófia Ádám	BME Budapest, Hungary
Raphaël Monat	Inria and University of Lille, France
Simmo Saan	University of Tartu, Estonia
Frank Schüssele	University of Freiburg, Germany

Benchmark Categories

Thomas Lemberger	LMU Munich, Germany

Infrastructure

Philipp Wendler	LMU Munich, Germany (BenchExec)
Po-Chun Chien	LMU Munich, Germany (BenchCloud)
Marek Jankola	LMU Munich, Germany (BenchCloud)
Henrik Wachowitz	LMU Munich, Germany (FM-Weck)
Matthias Kettl	LMU Munich, Germany (Competition Scripts)
Marian Lingsch-Rosenfeld	LMU Munich, Germany (WitnessLint)

Reproducibility

Levente Bajczi

BME Budapest, Hungary

Qualification

Paulína Ayaziová

Masaryk University, Brno, Czechia

Matthias Heizmann

University of Freiburg, Germany

Marian Lingsch-Rosenfeld

LMU Munich, Germany

Felix Mächtle

University of Luebeck, Germany

Raphaël Monat

Inria and University of Lille, France

Malte Mues

Bergische Universität Wuppertal, Germany

Jan-Niclas Serr

University of Luebeck, Germany

SV-COMP Program Committee and Jury

Dirk Beyer (Co-chair)

LMU Munich, Germany

Jan Strejček (Co-chair)

Masaryk University, Brno, Czechia

Zsófia Ádám

BME Budapest, Hungary

Paulína Ayaziová

Masaryk University, Brno, Czechia

Levente Bajczi

BME Budapest, Hungary

Max Barth

LMU Munich, Germany

Manuel Bentele

University of Freiburg, Germany

Martin Blichá

Charles University, Czechia

Lei Bu

Nanjing University, China

Zhenbang Chen

National University of Defense Technology,
China

Po-Chun Chien

LMU Munich, Germany

Tomáš Dacík

Brno University of Technology, Czechia

Paolo Di Biase

Unimol, Italy

Daniel Dietsch

University of Freiburg, Germany

Marcel Ebbinghaus

University of Freiburg, Germany

Gidon Ernst

LMU Munich, Germany

Fei He

Tsinghua University, China

Matthias Heizmann

University of Freiburg, Germany

Karoliine Holter

University of Tartu, Estonia

Omar Inverso

Gran Sasso Science Institute, Italy

Marek Jankola

LMU Munich, Germany

Martin Jonáš

Masaryk University, Brno, Czechia

Dominik Klumpp

University of Freiburg, Germany

Xianzhiyu Li

University of Manchester, UK

Marian Lingsch-Rosenfeld
 Nils Lommen
 Nils Loose
 Felix Mächtle
 Ravindra Metta
 Raphaël Monat
 Hassan Mousavi

Naïm Moussaoui Remil
 Malte Mues
 Hernán Ponce de León
 Ali Rasim Koçal
 Matthew Richards
 Simmo Saan
 Peter Schrammel
 Frank Schüssele
 Adéla Štěpková
 Csanád Telbisz
 Hiroshi Unno
 Vesal Vojdani
 Henrik Wachowitz
 Tong Wu
 Zuchao Yang
 Giacomo Zanatta

LMU Munich, Germany
 RWTH Aachen, Germany
 University of Luebeck, Germany
 University of Luebeck, Germany
 Tata Consultancy Services, India
 Inria and University of Lille, France
 University of Tehran and Tehran Institute for
 Advanced Studies, Iran
 Inria Paris and École Normale Supérieure, France
 Bergische Universität Wuppertal, Germany
 Huawei Dresden Research Center, Germany
 Technische Universität München, Germany
 University of New South Wales, Australia
 University of Tartu, Estonia
 Diffblue, UK
 University of Freiburg, Germany
 Masaryk University, Brno, Czechia
 BME Budapest, Hungary
 Tohoku University, Japan
 University of Tartu, Estonia
 LMU Munich, Germany
 University of Manchester, UK
 Xidian University, China
 Ca' Foscari University of Venice, Italy

Steering Committee

Dirk Beyer
 Dana Fisman
 Holger Hermanns
 Joost-Pieter Katoen (Chair)
 Kim G. Larsen
 Corina Păsăreanu

LMU Munich, Germany
 Ben-Gurion University, Israel
 Universität des Saarlandes, Germany
 RWTH Aachen, Germany
 Aalborg Universitet, Denmark
 NASA Ames, USA

Additional Reviewers

Miriam Ackermann
 Rajab Aghamov
 Amir Ahmadian
 Daneshvar Amrollahi

Jie An
 Ashwani Anand
 Elli Anastasiadi
 Bruno Andreotti

Roman Andriushchenko
Himanshu Arora
Nicola Assolini
Thom Badings
Daniel Baier
Nikhil Balaji
Ludovico Battista
Chanan Ben Tal
Zvika Berger
Martin Blich
Eline Bovy
Marco Bozzano
Konstantin Britikov
Asger Horn Brorholt
Francesca Cairol
Calvin Chau
Po-Chun Chien
Rachel Cleaveland
Robin Coutelier
Loris D'Antoni
Gabriel Dengler
Laurens Devos
Alastair Donaldson
Romina Doz
Kevin Dubrulle
Yizhak Elboher
Constantin Enea
Zainab Fatmi
Shenghua Feng
Mathias Fleury
Darius Foo
Markus Frohme
Maris Galesloot
Laura P. Gamboa Guzman
Xinyu Ge
Christina Gehnen
Enrico Ghiorzi
Juergen Giesl
Andrea Gimelli
Ehsan Goharshady
R Govind
Alberto Griggio
Thomas Hader
Edwin Hamel-de Le Court
Jonas Hansen

Ichiro Hasuo
Steeff Hegeman
Philippe Heim
Denghang Hu
Marieke Huisman
Antti Hyvärinen
Emilio Incerto
Shachar Itzhaky
Michael Jacks Jr
Marek Jankola
David N. Jansen
Nicolaj Østerby Jensen
Somesh Jha
Taylor T. Johnson
Aniruddha Joshi
Benjamin Lucien Kaminski
Jan-Christoph Kassing
Chantal Keller
Matthias Kettl
Åsmund Aqissiaq Arild Kløvstad
Sascha Klüppelholz
Paul Kobialka
Bram Kohlen
Tomas Kolarik
Igor Konnov
Merlijn Krales
Srdan Krstic
Konstantin Kueffner
Martin Kurečka
Robert Künnemann
Zbyněk Křivka
Hanna Lachnitt
Jean-Marie Lagniez
Min Lai
Engel Lefauchaux
Thomas Lemberger
Matthieu Lemerre
Moritz Leven Rosarius
Bohan Li
Renjue Li
Xie Li
Yong Li
Zhiyang Li
Shaokai Lin
Yi-Fan Lin

Marian Lingsch-Rosenfeld
Fang-Yi Lo
Nils Lommen
Alejandro Luque Cerpa
Filip Macák
Benedikt Maderbacher
Assia Mahboubi
Muhammad Mahmoud
Claude Marché
Jan Martens
Agustín E. Martínez-Suñé
Tomaz Mascarenhas
Kuldeep S. Meel
Jingyi Mei
Roland Meyer
Éléanore Meyer
Robert Modderman
Federico Mora
Harshit Jitendra Motwani
Vojtech Mrazek
Marco Muniz
Vít Musil
Alexander Nadel
Jasper Nalbach
Massimo Narizzano
Mahdi Nazeri
Grigory Neustroev
Sergei Novozhilov
Dirk Nowotka
Tommaso Padoan
Alasdair Paren
Julie Parreaux
Karim Pedemonte
David Perera
Elizaveta Pertseva
Annabell Petri
Ramchandra Phawade
Nir Piterman
Francesco Pontiggia
Danny Bøgsted Poulsen
Satya Prakash Nayak
Valentin Promies
Ritam Raha
Francesca Randone
Gianluca Redondi

Idan Refaeli
Andrew Reynolds
Alec Rosentrater
Nikolaj Rossander Kristensen
Diptarko Roy
Pedro Saccomani
Irmak Saglam
Mahmoud Salamati
Michael Sammler
Sriram Sankaranarayanan
Dominik Schmid
Jule Schmidt
Yannik Schnitzer
Morten Konggaard Schou
Peter Schrammel
Johann Schumann
Arijit Shaw
Riccardo Sieve
Simone Silveti
Nikhil Singh
Antonina Skurka
Mallku Soldevila
Sylvain Soliman
Gaëtan Staquet
Richard Stewing
Han Su
Yusen Su
Elina Sudit
Marnix Suilen
Amrita Suresh
Juraj Síf
Fouzi Tabouri
Rasmus Tollund
Stefano Tonetta
Wei-Lun Tsai
Max Tschaikowski
Nestan Tsiskaridze
Andrea Turrini
Viktor Vafeiadis
Niels van der Weide
Tom van Dijk
Mark van Wijk
Marcell Vazquez-Chanlatte
Freek Verbeek
Henrik Wachowitz

Zili Wang
Ziran Wang
Lennart Weingarten
Maximilian Weininger
Joshua Wendland
Philipp Wendler
Tim Willemse
Amalee Wilson
Thorsten Wißmann
Boris Wu
Hao Wu

Haoze Wu
Yu-Hsuan Wu
Akihisa Yamada
Mingqi Yang
Qiusong Yang
Wenzhang Yang
Robin Ziemek
Markel Zubia
Michal Šedý
Jakub Šárník

Contents

Software Verification

Verifying Floating-Point Programs in Stainless	3
<i>Andrea Gilot, Axel Bergström, and Eva Darulova</i>	
Efficient Verification of Lingua Franca Programs	23
<i>Peter Csaba Ölveczky, Mario Reja, Mikheil Rukhaia, Kyungmin Bae, and Mircea Marin</i>	
A Case Study in Firmware Verification: Applying Formal Methods to Intel® TDX Module	42
<i>Dirk Beyer, Po-Chun Chien, Bo-Yuan Huang, Nian-Ze Lee, and Thomas Lemberger</i>	
Extending FRET with SLEEC Rules: Formalization, Obligation Inference, and Monitoring	65
<i>Mahrokh Mirani, Paola Inverardi, Patrizio Pelliccione, Franco Raimondi, and Nicolas Troquard</i>	
Deciding Serializability in Network Systems	85
<i>Guy Amir, Mark Barbone, Nicolas Amat, and Jules Jacobs</i>	
VERISTRUCT: AI-assisted Automated Verification of Data-Structure Modules in Verus	109
<i>Chuyue Sun, Yican Sun, Daneshvar Amrollahi, Ethan Zhang, Shuvendu Lahiri, Shan Lu, David Dill, and Clark Barrett</i>	

Program Verification

On Deciding Constant Runtime of Linear Loops	131
<i>Florian Frohn, Jürgen Giesl, Peter Giesl, and Nils Lommen</i>	
Integrating String Reasoning in Symbolic Execution of C Programs	151
<i>Rachel Cleaveland and Clark Barrett</i>	
Same Engine, Multiple Gears: Parallelizing Fixpoint Iteration at Different Granularities	173
<i>Ali Rasim Kocal, Michael Schwarz, Simmo Saan, and Helmut Seidl</i>	

Gillian Debugging: Swinging Through the (Compositional Symbolic Execution) Trees	195
<i>Nat Karmios, Sacha-Élie Ayoun, and Philippa Gardner</i>	
RECHECK: Automated Contextual Improvement Verifier for Functional Calculi across User-Defined Operational Semantics	215
<i>Makoto Hamana and Kento Emoto</i>	
Data Structure Analysis for Binaries	235
<i>Sadra Bayat Tork, Nicholas Coughlin, Alicia Michael, James Tobler, and Kirsten Winter</i>	
Java Analysis	
jMT: Testing Correctness of Java Memory Models	257
<i>Lukas Panneke and Heike Wehrheim</i>	
Demonstrating ARG-V's Generation of Realistic Java Benchmarks for SV-COMP	277
<i>Charles Moloney, Robert Dyer, and Elena Sherman</i>	
Hardware Verification and Model Checking	
CTL* Model Checking on Infinite Families of Finite-State Labeled Transition Systems	289
<i>Roberto Pettinau and Christoph Matheja</i>	
ReVEAL: GNN-Guided Reverse Engineering for Formal Verification of Optimized Multipliers	309
<i>Chen Chen, Daniela Kaufmann, Chenhui Deng, Zhan Song, Hongce Zhang, and Cunxi Yu</i>	
EvolveGen : Algorithmic Level Hardware Model Checking Benchmark Generation through Reinforcement Learning	334
<i>Guangyu Hu, Xiaofeng Zhou, Wei Zhang, and Hongce Zhang</i>	
Verifying First-Order Temporal Properties of Infinite-State Systems via Timers and Rankings	353
<i>Raz Lotan, Neta Elad, Oded Padon, and Sharon Shoham</i>	
Revisiting Stateful Partial-Order Reduction	373
<i>Frédéric Herbreteau, Gérald Point, Gautham Viswanathan, and Igor Walukiewicz</i>	

Deconstructing Subset Construction: Reducing While Determinizing	394
<i>John Nicol and Markus Frohme</i>	

Quantum

Equivalence Checking of Quantum Circuits via Path-Sum and Weighted Model Counting	419
<i>Wei-Jia Huang, Christophe Chareton, Yu-Fang Chen, Kai-Min Chung, Min-Hsiu Hsieh, Alfons Laarman, and Jingyi Mei</i>	

Error-Tolerant Quantum State Discrimination: Optimization and Quantum Circuit Synthesis	440
<i>Chien-Kai Ma, Bo-Hung Chen, Tian-Fu Chen, Dah-Wei Chiou, and Jie-Hong R. Jiang</i>	

SV-COMP

Evaluating Software Verifiers for C, Java, and SV-LIB: (Report on SV-COMP 2026)	461
<i>Dirk Beyer and Jan Strejček</i>	

DASA: Fully Gradient-Based Program Analysis (Competition Contribution)	503
<i>Felix Mächtle, Jan-Niclas Serr, Nils Loose, and Thomas Eisenbarth</i>	

EMERGENTHETA: Experimental Analyses within the Theta Framework (Competition Contribution)	510
<i>Milán Mondok, Csanád Telbisz, Levente Bajczi, Dániel Kovács, Mihály Dobos-Kovács, and Vince Molnár</i>	

Goblint: A Portfolio for Mixed Flow-Sensitive Abstract Interpretation: (Competition Contribution)	516
<i>Simmo Saan, Ali Rasim Kocal, Michael Petter, Karoliine Holter, Julian Erhard, Michael Schwarz, Vesal Vojdani, and Helmut Seidl</i>	

GOBLITCH: Combining Abstract Interpretation with Symbolic Execution via Witnesses: (Competition Contribution)	523
<i>Karoliine Holter, Paulína Ayaziová, Simmo Saan, Jan Strejček, and Vesal Vojdani</i>	

Hornix: From LLVM IR to Constrained Horn Clauses and Back (Competition Contribution)	530
<i>Martin Blicha, Jan Kofroň, and Oliver Glitta</i>	

IEKKE: A SAT-Based Bounded-Round Verifier for Multi-Threaded Programs (Competition Contribution)	536
<i>Paolo Di Biase, Bernd Fischer, Salvatore La Torre, Peter Schrammel, and Gennaro Parlato</i>	
JLiSA: The Java Frontend of the Library for Static Analysis (Competition Contribution)	543
<i>Vincenzo Arceri, Luca Negrini, Giacomo Zanatta, Filippo Bianchi, Teodors Lisovenko, Luca Olivieri, and Pietro Ferrara</i>	
Mopsa-C: Towards Incorrectness and Termination Verdicts (Competition Contribution)	551
<i>Marco Milanese, Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné</i>	
RE3VER: Reverse and Verify: (Competition Contribution)	559
<i>Adéla Štěpková, Martin Jonáš, and Jan Strejček</i>	
REFUNCTION: Conditional Termination by Abstract Interpretation of Numerical C Programs: (Competition Contribution)	565
<i>Naïm Moussaoui Remil and Caterina Urban</i>	
SEAL: Symbolic Execution with Separation Logic: (Competition Contribution)	571
<i>Tomáš Brabec, Tomáš Dacík, and Tomáš Vojnar</i>	
SWAT: Improvements to the Symbolic Executor (Competition Contribution)	577
<i>Nils Loose, Florian Sieck, Felix Mächtle, and Thomas Eisenbarth</i>	
Symbiotic 11 Predicate Abstraction Joins the Party: (Competition Contribution)	583
<i>Paulína Ayaziová, Martin Jonáš, Vincent Mihalkovič, Jindřich Sedláček, and Jan Strejček</i>	
ULTIMATE AUTOMIZER with a One-Dimensional Memory Model: (Competition Contribution)	589
<i>Manuel Bentele, Max Barth, Marcel Ebbinghaus, Jan Körner, Daniel Dietsch, Matthias Heizmann, Dominik Klumpp, Frank Schüssele, and Andreas Podelski</i>	

Ultimate Paralyzer: Parallel Trace Abstraction (Competition Contribution)	595
<i>Max Barth, Daniel Dietsch, Matthias Heizmann, and Marie-Christine Jakobs</i>	
Author Index	601

Software Verification



Verifying Floating-Point Programs in Stainless

Andrea Gilot*✉, Axel Bergström*, and Eva Darulova

Uppsala University

{andrea.gilot,axel.bergstrom,eva.darulova}@it.uu.se

Abstract. We extend the Stainless deductive verifier with floating-point support, providing the first automated verification support for floating-point numbers for a subset of Scala that includes polymorphism, recursion and higher-order functions. We follow the recent approach in the KeY verifier to axiomatise reasoning about mathematical functions, but go further by supporting all functions from Scala’s math API, and by verifying the correctness of the axioms against the actual implementation in Stainless itself. We validate Stainless’ floating-point support on a new set of benchmarks sampled from real-world code from GitHub, showing that it can verify specifications about, *e.g.*, ranges of output or absence of special values for most supported functions, or produce counter-examples when the specifications do not hold.

Keywords: Floating-Point Arithmetic · Transcendental Functions · Deductive Verification

1 Introduction

Floating-point arithmetic is ubiquitous in modern software. While it is often discussed in the context of scientific computing, embedded systems, and machine learning, it is first and foremost pervasive in everyday user code [19]. Yet, verification tools do not always support floating-point reasoning, limiting their use in practice. For example, the absence of floating-point support in Liquid Haskell [41] has been identified as a limitation to its applicability in the real world [40].

This work introduces bit-accurate floating-point reasoning into Stainless [21], a deductive verification framework for Scala programs. Two main challenges in this integration are correctness and performance. To ensure correctness, the implementation must carefully address semantic discrepancies between the JVM and SMT-LIB, and the non-standard behaviour of equality.

Moreover, reasoning about floating-point arithmetic in SMT solvers is known to be expensive, requiring a careful encoding of verification conditions using specialised SMT-LIB constructs when available.

We start by reproducing the floating-point support recently introduced in KeY [1], demonstrating that this approach can be generalised to other verifiers. Our integration extends this support to the subset of the Scala language handled

* These authors contributed equally to this work.

by Stainless, enabling the verification of programs that combine floating-point computations with, among others, higher-order functions and algebraic data types, features unsupported in KeY. We address the challenge of combining polymorphic equality, often assumed to be reflexive, with floating-point reasoning, where this assumption does not hold (because $\text{NaN} \neq \text{NaN}$). Ignoring this mismatch can lead to unsoundness, as theorems proven for polymorphic types may be unsafely instantiated with floating-point types.

We implement a standard math library for Stainless that mirrors the Scala standard library API, and verify specifications *in Stainless itself* that were previously only axiomatised in KeY or not supported at all. While these specifications are partial, *e.g.*, encoding that `sin` returns a value between -1 and 1, they are sometimes sufficient for verifying user code with transcendental functions. SMT-solvers do not support transcendental functions, and inlining the implementation of math library functions will typically yield unacceptably long solving times.

To minimise user annotation effort, Stainless *automatically* generates verification conditions that capture subtle aspects of floating-point behaviour that are counter-intuitive and often misunderstood, even among experienced developers [14]. Specifically, it enforces non-`NaN` preconditions for floating-point comparisons and verifies the safety of casts between floating-point and integer types.

We evaluate Stainless on the benchmarks proposed by KeY [1], showing comparable support for floating-point reasoning. We further assess Stainless on 103 real-world floating-point functions mined from open-source GitHub repositories, identifying both practical use cases and remaining challenges—for Stainless and beyond—for achieving comprehensive floating-point verification support.

Our results show that Stainless successfully verifies all benchmarks from KeY and either verifies or produces counterexamples for 82% of user benchmarks. Reasoning about transcendental functions via properties verified in our mathematical library is sufficient for more than half of these user benchmarks, highlighting both its practical effectiveness but also its current limitations. Finally, automated checks for unintuitive floating-point behaviour detect every occurrence of the issues checked for in user code, effectively identifying potential bugs.

Contributions. In summary, we make the following contributions:

- We extend the Stainless verifier with floating-point support¹, the recent SMT-solver Bitwuzla [34] as a backend specialising in floating-point reasoning, and usability features such as automatic checks for unintuitive floating-point behaviour (Section 3).
- We implement a standard math library for Stainless that closely follows the OpenJDK math library implementation and verify mathematical properties of this library with Stainless itself (Section 4).
- We evaluate Stainless on the set of benchmarks proposed by KeY as well as on user-written floating-point code mined on GitHub² (Section 5).

¹ Available at: <https://github.com/epfl-lara/stainless>.

² <https://github.com/fxpl/stainless-float-benchmarks>

```

case class StormDay(moves: Int, errors: Int) {
  require(moves >= errors && errors >= 0 && moves > 0) // class invariant

  def accuracyPercent: Float = {
    100 * (moves - errors) / moves.toFloat
  }.ensuring(result => 0 <= result && result <= 100) // postcondition
}

def gradient(prediction: Double, label: Double): Double = {
  require(prediction.isFinite && label.isFinite) // precondition
  - 4.0 * label / (1.0 + math.exp(2.0 * label * prediction))
}.ensuring(result => !result.isNaN)

def limit(maxMagnitude: Double): Vector = {
  if (getMagnitude > maxMagnitude) // NaN check inserted here
    getNewMagnitude(maxMagnitude)
  else Vector(x, y)
}

```

Fig. 1: Example floating-point programs extracted from GitHub

2 Overview

Stainless did not previously support floating-point reasoning. In this section, we provide an overview of Stainless and of the floating-point support introduced in this work. Stainless is a contract-based verifier for Scala programs [21]. Every Stainless program is valid Scala code and can be compiled and executed on the JVM like any other Scala program. The class `StormDay` in Figure 1 illustrates a fragment of the Lichess codebase, an online chess platform. The function `accuracyPercent` computes a player’s accuracy by counting the number of correct moves in a sequence of chess moves. Under the assumption that the number of moves is strictly positive and greater than the number of errors (expressed in the class invariant in the **require** clause), we can specify in the postcondition (**ensuring** clause) that the result must lie within the interval $[0, 100]$.

During verification, Stainless generates verification conditions that are discharged by SMT solvers. In this case, Stainless reports the counterexample `StormDay(1073741832, 730144766)`, for which the function returns $-2.9586256E-5$, violating the postcondition. This result occurs because the left multiplication uses integer arithmetic, overflows to a negative value, and is then divided. Rewriting the body of `accuracyPercent` to `100 * ((moves - errors) / moves.toFloat)` forces the floating-point division to occur before the multiplication by 100, guaranteeing that the result always lies within the expected range.

The function `gradient` in Figure 1 implements a gradient computation in Apache Spark [3], a Scala framework for large-scale data processing and the most popular Scala library on GitHub. For this example, we specify that the inputs are finite, *i.e.*, they are neither of the floating-point special values $\pm\text{Infinity}$

nor `NaN`. One can obtain infinite floating-point values due to, *e.g.*, overflow and “Not a Number” values due to mathematically invalid operations, *e.g.*, `sqrt(-1.0)`. Stainless attempts to verify that the function never returns `NaN`, which one can view as saying that it does not encounter an error. When run on this snippet, Stainless reports a counterexample in which `label` is assigned a value of very large magnitude (causing overflow when multiplied by 2) and `prediction` is set to 0, leading to `Infinity * 0 = NaN`. Again reassociating the multiplications to the right solves the problem and ensures compliance with the specification, as verified by Stainless.

This example poses two challenges for the verifier: the detection of `NaN` values and the presence of the exponential function. Detecting `NaN` values is difficult for both developers and static analysers because their precise characterisation requires bit-level reasoning. The difficulty increases in the presence of transcendental functions, `math.exp` in this example, that are not natively supported by SMT solvers and whose implementations are typically too complex to be inlined.

Stainless supports reasoning about mathematical functions from the Scala standard library such as trigonometric, exponential or logarithmic functions. Following the approach of KeY [1], these functions are modelled as axiomatised uninterpreted symbols. This abstraction enables Stainless to perform basic reasoning about the occurrence of special values such as `NaN`, without requiring complete semantic modelling. Unlike KeY, we verify the axioms against the actual implementation with Stainless itself—stress-testing its floating-point support as these functions are highly numerical in nature; Section 4 provides more details.

Special-value bugs frequently arise from missing `NaN` checks [13]. For example, NumPy previously contained a bug in its `max` function, which returned an arbitrary number instead of the maximum when `NaN` values were present in the input array. The cause for such bugs is `NaN`’s unintuitive behaviour: any comparison involving `NaN` evaluates to false, including `NaN == NaN`. To prevent such errors from going undetected, Stainless automatically inserts `NaN` checks for all comparisons and equality operations, ensuring that neither operand is `NaN`.

The function `limit` in Figure 1 clamps the magnitude of a 2D vector if it exceeds a given threshold. If the input threshold is `NaN`, the comparison evaluates to false, causing the function to return the original vector, effectively ignoring the invalid input. With the inserted `NaN` checks, Stainless automatically detects such cases and reports a corresponding counterexample. When comparisons with `NaN` are intentional, users can explicitly disable these checks.

Stainless also inserts checks when casting floating-point values to integers that verify that input arguments are in the correct range. In Scala, casting `NaN` yields 0, while out-of-range values are clamped to the maximum or minimum representable 32-bit or 64-bit integer. For 16-bit and 8-bit integers, however, out-of-range values wrap around modulo the target range, introducing a further source of unintended behaviour.

Most of the properties checked for in our examples and evaluation benchmarks involve range analysis, sign determination, and detection of special values. Floating-point equalities, which Stainless can also reason about, are less common.

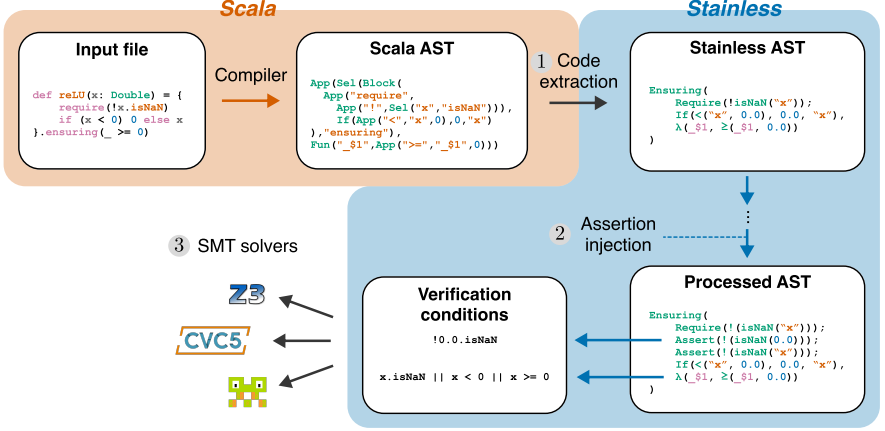


Fig. 2: Simplified architecture of Stainless

This is mainly because expressions that are equal over real numbers (in an ideal specification), will in general not be equal over floating-points due to rounding.

In addition to these high-level floating-point properties, Stainless can also reason about bit-level properties of floating-point values. Our case study in Section 4 requires reasoning about, for example, how bit-level operations on the exponents affect the corresponding floating-point values.

3 Implementation of Floating-Point Support in Stainless

We implement our extension within Stainless, which is implemented in Scala, and reuse existing infrastructure wherever possible. Figure 2 presents an overview of Stainless. Stainless operates on the ASTs produced by the Scala compiler. It then processes the AST in several phases including anti-aliasing, type encoding and termination measure inference. At the end of the pipeline, Stainless generates verification conditions and discharges them to SMT solvers. Note that the Scala compiler preserves floating points semantics (modulo compiler implementation bugs), and does not affect the verification results due to changes in floating-point rounding. Our extension reuses most of the existing infrastructure and brings modifications to only a few phases in the pipeline, thereby showing Stainless infrastructure’s modularity.

We now expand on the modifications we made to Stainless to support floating-point arithmetic both soundly and efficiently that we deem to be the most interesting for other tools and developers.

Code Extraction. (1) Part of the process consists in lowering Scala ASTs into Stainless ASTs. Scala provides high-level constructs, while Stainless uses an intermediate representation tailored for verification, which requires explicit encodings. By default, Stainless inlines non-recursive expressions and represents

primitive operations such as arithmetic, comparisons, and type casts with dedicated nodes. However, SMT-LIB defines more complex operators such as square root, min/max, and rounding (`ceil`, `floor`, `rint`). Mapping Scala calls directly to these operators reduces the size and complexity of verification conditions, so we disable inlining for such functions and translate them directly. For example, Scala implements the floating-point absolute value function by converting the input into binary, applying a mask, and reconstructing a floating-point value. Instead of inlining this bit-level implementation, we map calls to `abs` directly to the corresponding SMT-LIB operator.

Semantic discrepancies between the JVM and SMT-LIB also pose challenges. One such discrepancy concerns the modulo operator: in Scala `14.5 % 1.5` evaluates to `1.0`, while in SMT-LIB it yields `-0.5`. Correcting this difference (e.g., by adding an offset) introduces rounding errors, so Stainless does not currently support floating-point modulo. To assess the impact of this limitation, we analysed 30,421 floating-point functions mined from GitHub (Section 5.1). Only 108 (0.35 %) used modulo, typically with arguments 1 (to extract the fractional part) or 360 (for angle computations). In these cases, if one is willing to accept different floating-point rounding behaviour, one can replace `x % n` with `x - n * floor(x / n)` to avoid the modulo operation (although both expressions are mathematically equivalent over the reals, they accumulate rounding errors differently in floating-point arithmetic). Since Scala runs on the JVM, KeY is subject to the same limitation and likewise omits support for floating-point modulo.

A second discrepancy concerns the representation of `NaN` values. In IEEE 754, `NaN`s are encoded with all exponent bits set and one or more non-zero significand. This yields multiple encodings of `NaN` values where the varying significand bits form a payload. The JVM preserves these payloads, while SMT-LIB uses a single canonical `NaN` value. As a result, functions that expose the binary encoding of floating-point numbers diverge in semantics, since SMT-LIB does not differentiate between different binary `NaN` encodings. To avoid unsoundness, Stainless models binary conversion as an uninterpreted function whose result, when converted back to a floating-point value, equals the original floating-point value.

Equality and Polymorphism. The SMT-LIB standard distinguishes between bitwise equality and IEEE754 equality, where `+0` is equal to `-0` and `NaN` is not equal to anything, including itself. The non-reflexivity of equality is a problem when mixed with parametric polymorphism. In Stainless, reasoning about type variables relies on the assumption that equality is reflexive, symmetric, and transitive. When such type variables are instantiated with floating-point types or with user-defined types that override equality, these assumptions may lead to unsoundness. For instance, Stainless could incorrectly prove `NaN == NaN`.

One possible solution is to drop all assumptions about parametric equality and instead treat it as an uninterpreted symbol, as done in Flux [27]. Although this guarantees soundness, it breaks backward compatibility because many programs that previously verified in Stainless would no longer do so.

We adopt a more pragmatic approach. By default, Stainless only instantiates type variables with types whose equality satisfies reflexivity, symmetry, and

transitivity. If the user wishes to instantiate a polymorphic expression with `Float` or `Double`, they must annotate the type variable with `@noeq`. Stainless then treats equality on the annotated type variable as an uninterpreted predicate, thereby avoiding unsoundness due to assumed algebraic properties of equality.

Even disregarding `NaNs`, IEEE754 equality does not necessarily imply behavioural equivalence either. For example, `+0.0` and `-0.0` compare as equal, even though `.isPositive` returns true for the former and false for the latter. Stainless handles this behaviour correctly.

Assertion Injection. (2) One of the phases we most extensively modified in our work is assertion injection. For integer arithmetic, Stainless inserts checks by default to ensure that operations do not overflow and that divisions by zero do not occur. This behaviour can be disabled globally through command-line options or locally via wrappers. Assertions are injected during AST traversal, depending on the node being processed.

We extend this mechanism with additional checks for floating-point operations. In particular, we add `NaN`-checks on comparisons and safety checks for casts to integers. When processing a comparison node (including equality), Stainless now generates a verification condition ensuring that both operands are non-`NaN`. When traversing a type cast to an integer, Stainless generates two verification conditions: one ensuring that the value is not `NaN`, and another ensuring that the floating-point value lies within the integer domain. As demonstrated in our empirical evaluation (Section 5.4), these checks can detect real bugs in user code.

It could be argued that `NaN` values should be checked globally, *e.g.*, by disallowing them in any function argument. While appealing from a correctness perspective, such control would raise concerns about the usability of the tool. Excessive checks can overwhelm the SMT solver and substantially increase verification time. It is also unclear what assumptions these checks should make about the inputs of checked functions or expressions. In addition, `NaN` values may be acceptable in some contexts (*e.g.*, to describe out-of-domain behaviour). In such cases, users should retain the ability to disable these checks on a per-function basis. Another approach would be to provide fine-grained control, for example by supporting different levels of strictness or configurable options. Evaluating such usability trade-offs requires a user study, which we leave for future work.

Addition of Bitwuzla. (3) We extend the set of available SMT solvers in Stainless to include Bitwuzla [34]. Bitwuzla offers strong native support for floating-point arithmetic and bit-vectors, and outperforms other solvers on these types of queries (Section 5.3). However, it lacks support for algebraic data types (ADTs), which are heavily used in Scala programs. For verification conditions involving ADTs, Stainless falls back to general-purpose solvers such as Z3 [33] or cvc5 [5].

4 Case Study: Verified Standard Math Library

Floating-point programs often rely on standard mathematical libraries, particularly for transcendental functions [19]. However, implementations of these

functions are currently often too complex for SMT solvers to reason about directly. Instead, the KeY verifier for Java axiomatises them [1], exposing only selected properties such as their output range or behaviour on special input values. We adopt this property-based approach for Stainless, avoiding inlining low-level implementations and keeping verification tractable for programs that depend on them.

Stainless goes further than KeY by verifying that these exposed properties hold for implementations equivalent or close to the OpenJDK math library, which is based on the widely used FdLibm C library [39]. Specifically, we implement a verified mathematical library in Stainless that mirrors the Scala standard library API. Non-transcendental functions use the existing Scala implementation and are inlined during verification. We extend the set of supported transcendental functions beyond those axiomatised in KeY, covering the full Scala math library, including hyperbolic functions, logarithms, cube roots, and the hypotenuse. We also verify additional properties such as sign preservation and relations to the identity function. Table 3 in the appendix of the extended version of this paper [18] summarises and compares the properties supported in KeY and Stainless. Users can choose whether Stainless should reason about transcendental functions through these verified properties or whether it should inline their implementations.

For most transcendental functions (15 out of 18), the implementation in Stainless adapts FdLibm’s code into Single Static Assignment (SSA) form, in which each variable is assigned exactly once. This conversion eliminates mutability and allows us to bypass Stainless’s encoding of mutable state, which would otherwise inflate verification conditions and degrade solver performance. We perform the SSA transformation manually and validate the resulting code by testing it against the original implementation on ten million random inputs, stratified by magnitude, as well as on special values such as zeros, infinities, NaNs, and multiples of π . We also oversample subnormal numbers, as they have been shown to be a common source of bugs in floating-point code [13]. As Stainless contracts are compiled by default to runtime assertions, fuzzing the implementations also checks if the proven properties hold for the tested inputs.

Trigonometric Functions. The trigonometric functions `sin`, `cos`, and `tan` are implemented as two distinct steps: 1. a range reduction computing the input modulo $\pi/2$, and 2. the computation of the function value on the reduced argument. Properties of the second step can be verified using the same approach as for the other FdLibm functions. Some more extensive modifications were required to show that the first step returns a value in the expected range, due to poor SMT-solver performance.

The remainder modulo $\pi/2$ (`RemPio2`) operation has cases for infinite or NaN values, small inputs, intermediate inputs (magnitude $\sim \leq 2^{20}\pi/2$), and a final case that can handle all finite floating-point numbers. Stainless can easily handle the first two cases, but we did not manage to complete the proofs for the intermediate magnitude case due to poor SMT-solver performance. Instead, we remove the intermediate range case from our implementation, using the algorithm from the

final case also for intermediate inputs. We describe the algorithm for this final case and some of the modification we made for the proof.

It is not possible to compute the remainder using ordinary floating-point operations, since this would yield unacceptably low accuracy. Instead, the algorithm represents values using arrays of either floating-point values or integers to achieve greater precision than ordinary double-precision floats. After converting the input to this form, it is: 1. multiplied by a (input-magnitude-dependent) part of the expansion of $2/\pi$; 2. has the integer part removed; 3. is multiplied by a prefix of the expansion of $\pi/2$; and, 4. is compressed from an array to a double-precision float with a compensation term (another double-precision float containing the part of the result too small in magnitude to be represented by the first one). After our modifications, Stainless can show that this result is in the expected range of $[-\pi/4, \pi/4]$ in about 20 minutes.

Arithmetic operations on these arrays are implemented using loops. Since Stainless is a functional verifier, it does not support loops natively. Instead, Stainless converts loops to recursive functions that return a tuple containing the updated values of all variables modified by the loop. In SMT-LIB, these tuples are encoded as algebraic data types.

This means we cannot rely on the high performance of Bitwuzla, which does not support algebraic data types. Instead, we have to help `cvc5` solve the verification conditions by adding extra assertions and sometimes unrolling a few loop iterations. Both of these are done to improve solver performance. We also modify the conversion of the input to the array form, ensuring that the same array index always represents values of the same magnitude. This simplifies the loop invariants from a disjunction of different cases to a single case.

The original implementation uses an adaptive number of bits of the expansion of $2/\pi$, re-doing part of the computation with more bits if the precision is found to be insufficient. Since it would be very challenging, in Stainless, to show that this process terminates, we simplify the implementation by using a constant number of bits of $2/\pi$.

Finally, we also modified the loop implementing the compression in step 4 to admit a simple invariant. Our implementation of the loop is based on Algorithm 4.1 in Ogita et. al. [36], but uses the `FastTwoSum` function instead of the `twoSum` function. Together, these modifications allow us to show that the result is in the expected range, and therefore show that the trigonometric functions yield values in the expected ranges.

We tested the trigonometric functions on *all* single-precision floats and 10 billion random double-precision values, in addition to the fuzzing described earlier. Since the implementation is no longer bit-by-bit equivalent to the original, we have to modify the way we test the trigonometric functions. Stainless' implementation of trigonometric functions is within 1 unit in the last place (ulp) of the OpenJDK implementation on all tested inputs.

5 Evaluation

We evaluate our extension of Stainless on three sets of benchmarks to investigate the following research questions:

RQ1: How effectively does Stainless verify the correctness of real-world floating-point programs or generate counter-examples when verification fails?

RQ2: How does the performance of the supported SMT solvers compare on floating-point queries generated from real-world code?

RQ3: Do NaN-checks on comparisons and type cast safety checks detect possible bugs in real-world code?

5.1 Experimental Setup

We use three sets of benchmarks representing real-world floating-point programs: benchmarks used in the evaluation of KeY [1], functions from Stainless’ standard math library (Section 4), and floating-point functions extracted from open-source Scala projects on GitHub.

Throughout this section, we consider each annotated source *file* a single benchmark. Each such benchmark may contain multiple annotated *functions*, each of which may generate multiple verification conditions (VCs).

We run all experiments on Ubuntu 24.02 with an AMD Ryzen 5900X CPU (12 cores at a 3.7 GHz base clock) and 128 GB of memory. We use three SMT-solvers: Z3 [33] (version 4.13.0), cvc5 [5] (version 1.1.2), and Bitwuzla [34] (built from latest GitHub commit [35]). We run all solvers using a single thread.

For the KeY benchmarks, we disable Stainless’s usability-oriented NaN checks and safety checks on type casts, since KeY does not include these features. For the FdLibm-based benchmarks, we also disable these checks to avoid degraded solver performance due to the larger solver context caused by these checks. We keep these checks enabled for the empirical GitHub benchmarks.

KeY Benchmarks. Because Stainless primarily targets functional programs, we adapt the benchmarks to a more functional style. Specifically, we make classes immutable, rewrite mutating methods to return new instances, and replace small fixed-size arrays by tuples. We also merge tightly coupled Java source files into single Scala files, resulting in some Scala benchmarks corresponding to multiple classes or files in the original Java suite.

Unlike Stainless, the KeY verifier supports multiple contracts per method. We translated such KeY benchmarks by (1) adding preconditions common to all contracts as a precondition in Stainless, and (2) encoding each KeY contract as an implication in Stainless, combining them into a single conjunctive postcondition while omitting any duplicated preconditions. If a method in a KeY benchmark has multiple contracts, one of which is invalid, we avoid using the invalid contract to prove other contracts by generating two versions of the method: one containing only valid contracts and one including the invalid contract. No method in the KeY benchmarks contains more than one invalid contract.

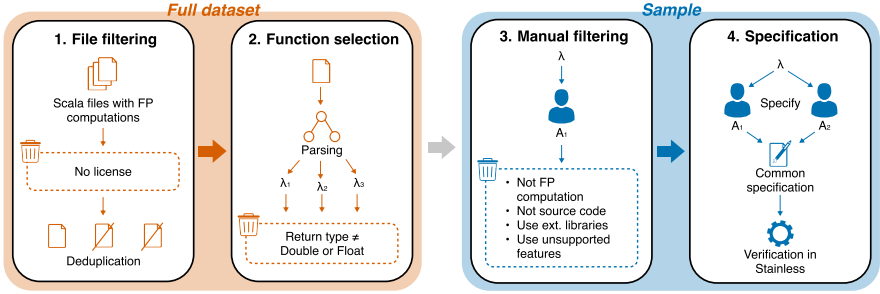


Fig. 3: Overview of the methodology to extract benchmarks from GitHub

FdLibm Benchmarks. The case study presented in Section 4 constitutes the second set of benchmarks. Each function in the math library is converted into one benchmark that requires proving all lemmas about the function, under the assumption that all lemmas about other functions in the library are correct. Two exceptions apply: the sine and cosine functions are combined into a single benchmark since they share most of their code, and the remainder-modulo- $\pi/2$ (RemPio2) operation—used to implement sine, cosine, and tangent functions—is treated as a standalone benchmark.

Empirical Benchmarks. Previous work [19] shows that common floating-point verification benchmarks do not accurately represent real-world programs. In their study, the authors extract a dataset of GitHub source files containing floating-point computations in statically typed languages. We build on this dataset and use the Scala subset to derive empirical benchmarks for Stainless. Figure 3 outlines our overall methodology for extracting and specifying these benchmarks.

We select all Scala files with open-source licenses, remove duplicates to mitigate large-scale code duplication [29], and extract all functions returning floating-point values. From these, we uniformly sample 500 functions and manually exclude:

- 178 functions that return a floating-point value but do not perform any floating-point computation (e.g., simple getters).
- 54 functions that do not belong to the source code of the repository (e.g., to a test suite).
- 97 functions that directly or indirectly call functions from external libraries (e.g., Breeze, Apache Spark).
- 70 functions that use language features unsupported by Stainless, such as macros, concurrency, or bounded polymorphism. When possible, we adapt such functions by removing unsupported features without altering their semantics.

After filtering, we obtain a set of 103 functions from 41 distinct repositories. Since functions may depend on one another, we group functions from the same source file and their dependencies into single verification benchmarks.

The first two authors independently write specifications for these functions, using in-code documentation, usage context, references to published algorithms, and semantic cues (e.g., a variable named `count` is assumed to be non-negative). We systematically examine whether the code can return special values, such as NaN, for valid inputs. The annotators then compare their specifications and resolve disagreements through discussion. When consensus cannot be reached, both specifications are retained and verified independently. If Stainless produces a counterexample when verifying one of the benchmarks, we manually correct the implementation and reattempt verification. This process yields two distinct benchmarks, one representing the original implementation and one incorporating the correction.

The resulting empirical benchmark suite consists of 79 benchmarks from 52 unique source files; 9 source files had unresolved specification disagreements, and 18 benchmarks were derived from fixing counterexamples.

5.2 RQ1: Verification Results on Full Benchmarks

We evaluate the floating-point support in Stainless by running end-to-end experiments on all benchmark sets. We configure Stainless to use `cvc5`, `Bitwuzla`, and `Z3` in a portfolio, with a 300-second timeout per VC for the KeY and the empirical benchmarks. We do not use any timeout for the more numerically intensive FdLibm benchmarks, as they are designed to establish the correctness of specific lemmas about the library functions, which requires all VCs to terminate. We do not empirically compare with KeY or Frama-C (e.g., measuring performance), as these tools target different programming paradigms. Such a comparison would require extensive modification of the benchmarks, as our translation of the KeY benchmarks demonstrates, making the results difficult to interpret.

Three benchmarks are excluded from the evaluation to prevent execution issues. Two of them are empirical benchmark: one causes stack-overflows in Stainless’ termination inference mechanism, while the other one causes Stainless to freeze, requiring manual interruption. We also exclude the FdLibm `RemPio2` benchmark, which triggers out-of-memory errors on the evaluation machine. The benchmark can, however, be successfully verified on the laptop used for the FdLibm case study, where it does not use an unusual amount of memory. We will investigate these issues further.

Table 1 summarises the evaluation results for the KeY and FdLibm benchmarks, and a subset of the empirical benchmarks. Comprehensive statistics for all empirical benchmarks are provided in Table 4 in the extended version of this paper [18]. For each benchmark, we report the minimum, median, and maximum Stainless end-to-end wall-clock execution times over five independent runs. We also include the total number of valid and invalid VCs, along with the numbers of VCs reported as unknown or resulting in timeouts. Additionally, for the median run, we report the number of valid and invalid VCs solved by each solver.

For the KeY and FdLibm benchmark sets, all VCs are successfully shown to be either valid or invalid. For the empirical benchmarks, Stainless reports at least one unknown result or timeout in 6 out of 79 empirical cases: 3 benchmarks experience

Table 1: Summary of five end-to-end runs. Min/median/max report end-to-end execution times. #VCs gives the number of verification conditions per benchmark with valid(/invalid/unknown/timeout) results in the run with the median execution time. cvc5, Bitwuzla, and Z3 report the number of VCs proven valid(/invalid) by each solver in that run. #fns shows the number of transcendental function calls (when nonzero).

(a) KeY benchmarks (300s timeout)

benchmark	min	median	max	#vcs	cvc5	Bitwuzla	Z3	#fns
CartesianPolar	12.92	13.05	13.15	3	3	0	0	3
Complex	94.74	95.62	96.06	9/1/0/0	9/1	0	0	3
FPLoop	4.48	4.55	4.67	30/3/0/0	13	12/3	5	
Matrix2	2.50	2.53	2.61	1	1	0	0	
Matrix3	129.44	130.34	130.65	1/1/0/0	1/1	0	0	
PineBench	234.66	245.32	247.01	105/14/0/0	58/2	46/12	1	
Rectangle	10.53	10.55	10.72	1/1/0/0	0/1	0	1	
Rotation	21.05	21.17	21.30	6	4	2	0	

(b) FdLibm benchmarks (no timeout)

benchmark	min	median	max	#VCs	cvc5	Bitwuzla	Z3	#fns
Acos	33.72	35.60	42.28	11	1	6	4	
Asin	105.46	171.82	859.33	8	1	6	1	
Atan	62.43	64.44	70.88	10	0	8	2	
Atan2	8.40	8.47	8.56	28	0	7	21	2
Cbrt	667.36	913.66	1178.50	11	1	7	3	
Cosh	40.07	41.85	45.40	9	1	7	1	4
Exp	49.85	52.04	152.58	16	0	11	5	
Expm1	3224.93	3969.72	4189.57	24	1	19	4	
Hypot	16.88	17.49	18.24	7	0	6	1	
Log	140.12	206.87	741.37	16	1	14	1	
Log10	4.10	4.21	4.22	6	0	4	2	1
Log1p	22.78	120.52	249.31	16	1	13	2	
Pow	9.40	9.57	9.63	24	1	21	2	
SinCos	194.80	220.15	511.64	42	12	3	27	
Sinh	111.82	119.59	125.88	9	1	8	0	3
Tan	17.80	17.93	18.09	13	3	4	6	
Tanh	4.35	4.42	4.59	6	0	5	1	2

(c) Subset of empirical benchmarks uniformly selected by median time (300s timeout)

benchmark	min	median	max	#VCs	cvc5	Bitwuzla	Z3	#fns
HTMLComp.Position_OK	3.30	3.35	3.43	2	1	1	0	
StreamingIO_CE	3.58	3.60	3.65	2/1/0/0	1/1	1	0	
BoundingBox_a1_OK	3.85	3.89	4.00	4	2	0	2	
AreaUnderCurve_OK	4.16	4.21	4.28	11	10	0	1	
DataHelper_OK	4.69	4.79	4.88	21	4	0	17	
Player_a0_OK	6.26	6.38	6.54	21	16	0	5	
Border_OK	7.74	7.83	7.87	77	14	0	63	
Mediator_a1_OK	9.41	9.50	9.71	56	10	0	46	
Variance_CE	15.25	15.44	15.61	37/2/0/0	18/2	4	15	
Vector2_a0_CE	33.42	33.55	33.75	7/2/0/0	3/2	0	4	
Common_TO	727.42	748.05	916.92	46/0/1/2	15	5	26	1

Table 2: Number of verification conditions solved by each solver

	#VCs	cvc5	Z3	Bitwuzla	Any solver
KeY	176	176 (100%)	150 (85.2%)	118 (67.0%)	176 (100%)
FdLibm	256	227 (88.7%)	200 (78.1%)	217 (84.8%)	254 (99.2%)
Empirical	1600	1595 (99.7%)	1528 (95.5%)	489 (30.6%)	1596 (99.8%)

at least one unknown result, and 4 encounter at least one timeout. All VCs are successfully shown to be either valid or invalid for 73 out of 79 empirical benchmarks.

Manual inspection reveals that for 8 empirical benchmarks detected as invalid by Stainless, the reported counterexamples are spurious.³ These benchmarks contain transcendental functions, which Stainless treats as opaque. That is, only their contract is available to the rest of the codebase and its implementation is hidden. The function call is translated to the SMT solver as an application of an uninterpreted function constrained solely by the contract. If the contract is not strong enough to discharge a verification condition, the solver can construct a model that satisfies the contract while violating the intended semantics of the function, leading to a spurious counterexample. Among the other 11 empirical benchmarks using transcendental functions, one yields a timeout, 5 are verified, and 5 yield non-spurious counterexamples. Overall, the axiomatic approach introduced by KeY is insufficient for 8 out of 19 benchmarks using transcendental functions, *i.e.*, a significant fraction.

Answer to RQ1: Our results indicate that Stainless is usually able to successfully verify or produce counterexamples for self-contained real-world floating-point programs (82 % of empirical cases). In some cases, we observe long verification times with a high variance, especially for the more numerically intensive FdLibm benchmarks. Transcendental functions still pose a challenge: the lemmas introduced in the case study over-approximate their behaviour, leading to spurious counterexamples in several benchmarks (8 out of 19).

5.3 RQ2: Comparison of Solvers on Verification Conditions

To evaluate and compare the performance of the SMT solvers supported by Stainless, we configure Stainless to generate one SMT-LIB file per VC in each benchmark. We run each solver on each generated SMT-LIB file five times and measure the total execution time. For VCs generated from the KeY and FdLibm benchmarks, we use a timeout of 300 seconds. Given the large number of empirical benchmarks, we use a reduced timeout of 120 seconds for this set.

Table 2 reports the number of VCs solved by cvc5, Z3 and Bitwuzla across all benchmark categories. The table also lists the total number of VCs solved by at least one of the three solvers. Compared to the end-to-end evaluation, we

³ For one particular run, the SMT-solvers are not guaranteed to produce the same counterexamples in different runs.

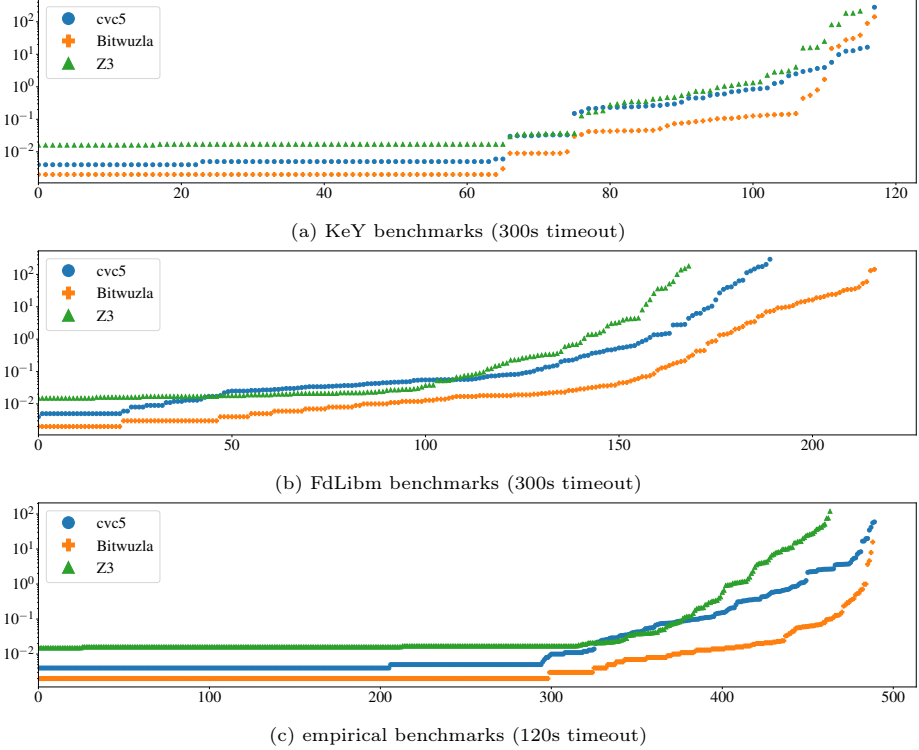


Fig. 4: Solver runtime for verification conditions supported by all solvers

observe fewer unsolved VCs, since unknown results can originate from Stainless itself; in such case we do not observe any unknown results from an SMT-solver. The results indicate that the portfolio approach is effective, with almost all VCs being solved by at least one solver.

Most unsolved VCs for Bitwuzla correspond to errors rather than performance limitations: it yields errors for 58 KeY benchmarks, 37 FdLibm benchmarks, and 1107 empirical benchmarks while *cvc5* and Z3 do not encounter any errors. We believe all of these errors are due to a lack of support for algebraic data types and the theory of integers. We therefore restrict the comparison between the solvers to the subset of VCs where Bitwuzla does not encounter any errors. Figure 4 visualises the performance of the three solvers on this subset, and Figure 5 in the extended version of this paper [18] contains corresponding plots for all VCs.

Answer to RQ2: We get a clear ranking of the solvers for the cases where Bitwuzla does not encounter any errors: Bitwuzla performs better than *cvc5*, especially for the numerically intensive FdLibm benchmarks, and *cvc5* performs better than Z3. Since Bitwuzla does not support algebraic data types or the theory of integers, *cvc5* still solves more VCs overall.

5.4 RQ3: Impact of NaN-Checks and Type Cast Safety Checks

Among the 103 functions in our empirical benchmark set, 9 contain floating-point comparisons. In all these cases, proper handling of NaN values is missing. In two functions, this omission causes the implementation to return a valid output even when the input is NaN, effectively silently ignoring NaN inputs. In one case the code comments explicitly assume that a false comparison implies that the input is valid and within a certain range, a property that does not hold when the input is NaN; we show this function in Figure 6 in the extended version of this paper [18].

In most of the functions, function arguments are compared against constant thresholds. In such cases, the assumption that inputs are non-NaN is sufficient to guarantee the absence of erroneous comparisons. When a function is lacking this assumption, Stainless successfully detects and reports potential NaN comparisons.

Floating-point to integer casts occur in two functions within the benchmark set. Both are unsafe, and are reported by Stainless with concrete counterexamples that trigger either NaN-to-integer conversions or out-of-bounds integer casts.

Answer to RQ3: Automated NaN-checks and type-cast safety checks effectively identify floating-point-related bugs in user code. In our empirical benchmark set, Stainless successfully detects all unsafe type casts and potential NaN comparisons, generating concrete counterexamples for each such case.

6 Related Work

To the best of our knowledge, we are the first to introduce floating-point reasoning into a deductive verifier for Scala. Stainless’ floating-point support via verification conditions discharged by SMT solvers using the floating-point SMT-LIB theory [37] follows the support available in KeY for Java, Why3 [15] for C [11] and Ada [16], and the Boogie intermediate language [28]. Compared to these tools, Stainless provides floating-point support for an input language with a different set of features that include polymorphism, recursion, higher-order functions and class invariants. It also adds automated NaN checks, reducing the annotation burden compared to existing verifiers. KeY introduced an axiomatic treatment of some transcendental functions from the standard math library [1]. In comparison, Stainless covers the entire standard math library and, for most functions, is able to verify the axioms against the actual implementation itself. For other JVM-based languages, deductive verifiers provide only limited support: VeriFast [25] treats floating-point operations as reals, while OpenJML [10] and LiquidJava [17] can parse floating-point programs but do not reason about them.

Abstract interpretation-based techniques [9, 26] can in principle automatically prove the absence of special values in floating-point code. However, applying these analyses effectively to real-world programs typically requires significant user tuning, such as selecting abstract domains or widening thresholds [7]. Ariadne [6] uses a combination of symbolic execution, real-valued SMT solving and testing to find inputs that trigger floating-point exceptions, including overflow and invalid oper-

ations. Our work subsumes this line of analysis: Stainless not only generates counterexamples through SMT solving, but also proves the absence of such exceptions.

Besides showing the absence of special values, recent research has developed static analyses to bound floating-point rounding errors [12, 20, 30, 32, 38]. These analyses, however, currently target small arithmetic kernels expressed in domain-specific languages, and lack support for features of mainstream programming languages such as user-defined data types, objects, and recursion.

Stainless verifies the axioms for the standard math library functions against the implementations using its own floating-point support. Bagnara et al. [4] verify properties of functions in the `math.h` C library, such as piecewise monotonicity and the absence of NaN results under valid inputs using an alternative approach that combines symbolic execution, abstract interpretation, and exhaustive testing. A complementary line of research verifies the error bounds for floating-point implementations of transcendental functions in interactive theorem provers (*e.g.*, HOL, Rocq) [2, 8, 22–24, 31]; such proofs require an extensive mathematical apparatus and significant user expertise and interaction. Harrison reports that the manual effort required for each such proof can vary from weeks to months [22].

7 Conclusion

Supporting floating-point reasoning in a deductive verifier for programs that use advanced language features such as classes, polymorphism, and higher-order functions requires specific restrictions, for example by limiting instantiations of polymorphic equality or by weakening equality through uninterpreted functions. With our extension of Stainless, we reproduced verification results of KeY and extended them by verifying a large subset of the Scala standard math library, which KeY previously axiomatised or did not support, and showed that Stainless supports verification of user code sampled from GitHub.

Future work includes detecting and eliminating spurious counterexamples arising from transcendental functions. When spurious examples are detected (*e.g.*, by evaluating the program), one possible strategy is to inline the full function implementation and reattempt verification. Current SMT solvers, however, are generally not able to reason efficiently about the complete implementation of transcendental functions, making this approach often ineffective in practice. Strengthening the contracts with additional properties can mitigate the problem, although it does not eliminate the underlying issue completely. Future work also includes investigating usability trade-offs, for example by adding further automated checks and designing more readable specification mechanisms.

Acknowledgments. Co-funded by the European Union (ERC, HORNET, 101163629). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

8 Data Availability Statement

Our floating-point support for Stainless is available in the main branch of the Stainless repository: <https://github.com/epfl-lara/stainless>. We also provide a Zenodo artifact (<https://doi.org/10.5281/zenodo.17486575>) containing the implementation, all benchmarks used in this paper, the collected evaluation results, and scripts to reproduce the experiments.

References

1. Abbasi, R., Schiffl, J., Darulova, E., Ulbrich, M., Ahrendt, W.: Combining Rule- and SMT-based Reasoning for Verifying Floating-Point Java Programs in KeY. *International Journal on Software Tools for Technology Transfer* **25**(2), 185–204 (2023). <https://doi.org/10.1007/s10009-022-00691-x>
2. Akbarpour, B., Abdel-Hamid, A.T., Tahar, S., Harrison, J.: Verifying a Synthesized Implementation of IEEE-754 Floating-Point Exponential Function using HOL. *The Computer Journal* **53**(4), 465–488 (2009). <https://doi.org/10.1093/comjnl/bxp023>
3. Apache Software Foundation: Apache Spark. <https://github.com/apache/spark> (2025), accessed: 2025-10-14
4. Bagnara, R., Chiari, M., Gori, R., Bagnara, A.: A Practical Approach to Verification of Floating-Point C/C++ Programs with math.h/cmath Functions. *ACM Trans. Softw. Eng. Methodol.* **30**(1) (2021). <https://doi.org/10.1145/3410875>
5. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2022). https://doi.org/10.1007/978-3-030-99524-9_24
6. Barr, E.T., Vo, T., Le, V., Su, Z.: Automatic Detection of Floating-Point Exceptions. In: *Proceedings of the Symposium on Principles of Programming Languages (POPL)* (2013). <https://doi.org/10.1145/2429069.2429133>
7. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A Static Analyzer for Large Safety-Critical Software. In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)* (2003). <https://doi.org/10.1145/781131.781153>
8. Boldo, S., Filliâtre, J.C., Melquiond, G.: Combining Coq and Gappa for Certifying Floating-Point Programs. In: *Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus)*, Held as part of Conference on Intelligent Computer Mathematics (CICM) (2009). https://doi.org/10.1007/978-3-642-02614-0_10
9. Chen, L., Miné, A., Cousot, P.: A Sound Floating-Point Polyhedra Abstract Domain. In: *Programming Languages and Systems (APLAS)* (2008). https://doi.org/10.1007/978-3-540-89330-1_2
10. Cok, D.R.: OpenJML: JML for Java 7 by Extending OpenJDK. In: *NASA Formal Methods* (2011). https://doi.org/10.1007/978-3-642-20398-5_35
11. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C - A Software Analysis Perspective. In: *International Conference on Software Engineering and Formal Methods (SEFM)* (2012). https://doi.org/10.1007/978-3-642-33826-7_16

12. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2018). https://doi.org/10.1007/978-3-319-89960-2_15
13. Di Franco, A., Guo, H., Rubio-González, C.: A Comprehensive Study of Real-World Numerical Bug Characteristics. In: International Conference on Automated Software Engineering (ASE) (2017). <https://doi.org/10.1109/ASE.2017.8115662>
14. Dinda, P., Hetland, C.: Do Developers Understand IEEE Floating Point? In: International Parallel and Distributed Processing Symposium (IPDPS) (2018). <https://doi.org/10.1109/IPDPS.2018.00068>
15. Filliâtre, J.C., Paskevich, A.: Why3 — Where Programs Meet Provers. In: European Symposium on Programming (ESOP), Programming Languages and Systems (2013). https://doi.org/10.1007/978-3-642-37036-6_8
16. Fumex, C., Marché, C., Moy, Y.: Automating the Verification of Floating-Point Programs. In: Verified Software. Theories, Tools, and Experiments (VSTTE) (2017). https://doi.org/10.1007/978-3-319-72308-2_7
17. Gamboa, C., Canelas, P., Timperley, C., Fonseca, A.: Usability-Oriented Design of Liquid Types for Java. In: International Conference on Software Engineering (ICSE) (2023). <https://doi.org/10.1109/ICSE48619.2023.00132>
18. Gilot, A., Bergström, A., Darulova, E.: Verifying Floating-Point Programs in Stainless. CoRR **abs/2601.14059** (2026). <https://doi.org/10.48550/arXiv.2601.14059>
19. Gilot, A., Wrigstad, T., Darulova, E.: A Large-Scale Study of Floating-Point Usage in Statically Typed Languages. CoRR **abs/2509.04936** (2025). <https://doi.org/10.48550/ARXIV.2509.04936>
20. Goubault, E., Putot, S.: Static Analysis of Finite Precision Computations. In: Verification, Model Checking, and Abstract Interpretation (VMCAI) (2011). https://doi.org/10.1007/978-3-642-18275-4_17
21. Hamza, J., Voirol, N., Kunčák, V.: System FR: Formalized Foundations for the Stainless Verifier. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). <https://doi.org/10.1145/3360592>
22. Harrison, J.: Floating Point Verification in HOL Light: The Exponential Function. Formal Methods in System Design **16**(3), 271–305 (2000). <https://doi.org/10.1023/A:1008712907154>
23. Harrison, J.: Formal Verification of Floating Point Trigonometric Functions. In: Formal Methods in Computer-Aided Design (FMCAD) (2000). https://doi.org/10.1007/3-540-40922-x_14
24. Harrison, J.: Formal Verification of Square Root Algorithms. Formal Methods in System Design **22**(2), 143–153 (2003). <https://doi.org/10.1023/A:1022973506233>
25. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: NASA Formal Methods (2011). https://doi.org/10.1007/978-3-642-20398-5_4
26. Jeannet, B., Miné, A.: Apron: A Library of Numerical Abstract Domains for Static Analysis. In: Computer Aided Verification (CAV) (2009). https://doi.org/10.1007/978-3-642-02658-4_52
27. Lehmann, N., Geller, A.T., Vazou, N., Jhala, R.: Flux: Liquid Types for Rust. Proc. ACM Program. Lang. **7**(PLDI) (2023). <https://doi.org/10.1145/3591283>
28. Leino, K.R.M.: This is Boogie 2. <https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/> (2008), accessed: 2025-10-14
29. Lopes, C.V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajjani, H., Vitek, J.: DéjàVu: A Map of Code Duplicates on GitHub. Proc. ACM Program. Lang. **1**(OOPSLA) (2017). <https://doi.org/10.1145/3133908>

30. Magron, V., Constantinides, G., Donaldson, A.: Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Trans. Math. Softw.* **43**(4) (2017). <https://doi.org/10.1145/3015465>
31. Melquiond, G.: Floating-point arithmetic in the Coq system. *Information and Computation* **216**, 14–23 (2012). <https://doi.org/10.1016/j.ic.2011.09.005>, special Issue: 8th Conference on Real Numbers and Computers
32. Moscato, M., Titolo, L., Dutle, A., Muñoz, C.A.: Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In: *Computer Safety, Reliability, and Security (SAFECOMP)* (2017). https://doi.org/10.1007/978-3-319-66266-4_14
33. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008). https://doi.org/10.1007/978-3-540-78800-3_24
34. Niemetz, A., Preiner, M.: Bitwuzla. In: *Computer Aided Verification (CAV)* (2023). https://doi.org/10.1007/978-3-031-37703-7_1
35. Niemetz, A., Preiner, M.: Bitwuzla public repository - commit used for experiments. <https://github.com/bitwuzla/bitwuzla/tree/929af621f6fe65c78236df70b8eb9343a5914d9f> (2025), accessed: 2025-10-14
36. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. *SIAM journal on scientific computing* **26**(6), 1955–1988 (2005). <https://doi.org/10.1137/030601818>
37. Rümmer, P., Wahl, T.: An SMT-LIB theory of binary floating-point arithmetic. In: *International Workshop on Satisfiability Modulo Theories (SMT)* (2010)
38. Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. *ACM Trans. Program. Lang. Syst.* **41**(1) (2018). <https://doi.org/10.1145/3230733>
39. Sun Microsystems Inc.: FDLIBM Version 5.3. <https://www.netlib.org/fdlibm/> (2022), accessed: 2025-10-14
40. Vazou, N., Seidel, E.L., Jhala, R.: LiquidHaskell: Experience with Refinement Types in the Real World. In: *Symposium on Haskell* (2014). <https://doi.org/10.1145/2633357.2633366>
41. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement Types for Haskell. In: *International Conference on Functional Programming (ICFP)* (2014). <https://doi.org/10.1145/2628136.2628161>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Efficient Verification of Lingua Franca Programs

Peter Csaba Ölveczky¹, Mario Reja², Mikheil Rukhaia³, Kyungmin Bae⁴,
and Mircea Marin²

¹ University of Oslo, Oslo, Norway

² West University of Timișoara, Timișoara, Romania

³ Institute of Applied Mathematics, Tbilisi State University, Tbilisi, Georgia

⁴ Pohang University of Science and Technology, Pohang, South Korea

Abstract. We present an executable rewriting logic semantics for the LINGUA FRANCA (LF) coordination language for cyber-physical systems developed at UC Berkeley. Our semantics enables automatic formal analysis of LF programs using Maude and Real-Time Maude. We capture the intended “logical-time” semantics of LF, which gives us correct analysis that is also magnitudes faster than other verification approaches for LF, which are based on an interleaving semantics. We capture larger subsets of LF than those other approaches, and support all analysis methods provided by them. Benchmarking on the LF VERIFIER benchmark suite shows that our analyses drastically outperform those of LF VERIFIER. We have also extended the LF compiler to automatically generate a Maude model from an LF program and to perform the desired analysis.

1 Introduction

LINGUA FRANCA (LF) [12,14,10] is a coordination language for cyber-physical systems, developed by the group of Edward A. Lee at UC Berkeley, that provides *deterministic concurrency*: the only source of nondeterminism are *external actions*.

In LF, *reactions* execute programs when triggered. Reaction executions are considered logically instantaneous, so that many reactions can be executed at the same logical time. For example, one reaction’s output may trigger another reaction’s execution *at the same logical time*. LF has a *logical-time* (or “discrete-event”) semantics ([12, pp. 7 and 20], [9, p. 4], [14, pp. 5, 20, and 24]): *all* reactions executed at the same logical time can be seen to happen in *one* step.

The goal of our paper is to formalize this intended semantics of LF and to support automatic formal analysis of LF programs based on this semantics; these are known to be challenging tasks: “*formally modeling the behavior of [discrete-event systems] using a conventional operational model [...] is difficult because one needs to specify properties globally over execution traces, a task that cannot be easily achieved at the transition level*” [9, p. 2].

We are aware of three efforts providing formal verification for LF, which all fall short of capturing its discrete-event semantics or covering realistic programs:

1. Sirjani, Lee, and Khamespanah [22] translate three LF programs into an extension of Timed Rebeca [1] to perform reachability analysis.

2. In [9], the UC Berkeley group encodes *bounded* model checking of a subset of LF as an SMT problem. Their LF VERIFIER tool can model check a fragment of metric temporal logic, so that it is sufficient to consider behaviors up to time $h(\phi)$ for a given MTL formula ϕ .
3. In [13], we provide a first proof-of-concept rewriting logic [15] semantics for LF and some basic Maude [6] analysis for LF programs.

Efforts (1) and (2) provide an “event-based” semantics, where each reaction execution corresponds to a step (“In the proposed axiomatic semantics, a *reaction invocation* defines a transition, which matches the *event-based semantics* in Sirjani et al.” [9, p. 8]). These efforts do not cover external events (“physical actions”), so they model check what LF guarantees are deterministic models. Not only do they not capture the intended logical-time semantics of LF, but, since they cover all interleavings in a “big” (synchronous) step, their verification is very slow.

Effort (3) captures the logical-time semantics of LF, but does not support: (i) reactions with multiple triggers, (ii) analyzing *timed* temporal logic properties, which are the ones analyzed in [9], or (iii) *dataflow* (or “action”) properties, which are central in [9]. We therefore could not cover most systems and desired properties in the LF VERIFIER benchmark suite in [13].

In this paper we build on the work in [13] to provide the desired logical-time semantics for a larger subset of LF than supported by other verification efforts, including *both* nondeterministic physical actions and multiple reaction triggers, and to provide many more analysis methods than other such efforts. We support both unbounded and time-bounded simulation, reachability analysis, LTL model checking, and timed CTL model checking, where the basic properties can be both state properties, dataflow properties, and “mixed” properties. By cleverly handling the tags in the event queues, the reachable state spaces in our executions in many cases remain finite, so that we have terminating *unbounded* reachability analysis and LTL and timed CTL model checking, as shown in Section 4.

In contrast to [13], we have also extended the LF compiler to (i) automatically generate a Maude model from an LF program, and (ii) to automatically perform the Maude analysis of the LF program, annotated with properties to verify, so that the LF user can analyze her program without any knowledge of Maude.

We give background to LF and Maude in Section 2, and formalize the semantics of LF in Section 3. We demonstrate a broad range of formal analyses of LF programs in Section 4, and benchmark all LF VERIFIER benchmarks with their desired properties in Section 5. Our analyses drastically outperform those in [9]: they take less than 0.1 seconds, whereas the analysis in [9] can take more than 20 minutes. We introduce our LF-MC tool for automatically verifying LF programs in Section 6, and discuss other related work in Section 7.

We refer to our longer report [20] for more detail, and to <https://the-mrd.github.io/tacas-artifact> for our executable formal semantics and analysis tool LF-MC for LF, with all examples and benchmarks.

2 Preliminaries

Lingua Franca. LINGUA FRANCA (LF) [12,14,10] is a coordination language for cyber-physical systems that supports *reactor-oriented* programming [11]. A system is composed of *reactors*, each containing state variables, ports, actions, timers, and an ordered list of *reactions* which are invoked in response to *triggers*. A trigger can be an input port, a timer, or an action. Timers generate periodic events. Logical actions schedule future events with delay. *Physical actions* represent external events whose timing and values are outside program control.

A *reaction* has the form $\text{reaction}(\text{triggers}) \rightarrow \text{effects} \{= \text{body} =\}$, where *triggers* is a set of triggers, *effects* is the ports and actions that the reaction *may* write to or schedule, and *body* specifies the reaction’s behavior in a *target language* (such as C and Python; hence “Lingua Franca”). A reaction is invoked when at least one of its triggers is present, *but* can execute only when *all triggers* are known to be present or absent at this logical time (see below). Depending on the values of state variables and inputs/actions, a reaction invocation may or may not produce effects. Reaction executions are considered *logically instantaneous*.

Ports are connected by *immediate* or *delayed connections*. An event travels *logically instantaneously* from an output port to the input ports connected via immediate connections. Events are time-stamped with tags $\langle t, m \rangle$ consisting of a *time value* t and a *micro-step* index $m \in \mathbb{N}$. An event may also carry a value. LF distinguishes between events from the physical environment and events produced by the program by considering two time lines: *logical time* and *physical time*.

Since reaction executions (or “invocations”) are logically instantaneous, a reaction may produce an output that travels along an *immediate* connection and triggers another reaction *at the same logical time*, whose execution could trigger yet another reaction at this same logical time, and so on. LF has a *logical-time* semantics: all these reactions execute at the same logical time in one “big step.”

Reactions triggered *at the same logical time* must satisfy: (i) if a reaction r produces an output that immediately triggers r' , then r must execute before r' ; and (ii) reactions in the *same reactor* execute in their order of declaration (to ensure determinism and mutual exclusion, since they access the same state variables). These constraints define a directed graph, called the *acyclic precedence graph* (APG) [14], with reactions as nodes and the above constraints as edges; any such APG must be acyclic for a program to be a *valid* LF program.

Example 1. Figures 1 and 2 show an LF program from [12]. It has three reactors y , r , and x , and only immediate connections. The white circle is the logical action startup, which triggers at startup. Since all connections are immediate, *all* reactions (depicted as chevrons) are triggered at the same logical time by startup. Their execution must satisfy the constraints in the APG in Figure 2; the execution of reaction **2** must wait until reaction **1** has finished executing. Although it does not happen in this example, *if* some invocations of the reaction in r produce no output, the execution of **2** cannot wait for **1** to finish. $\square$

```

reactor X {
  input dbl:int;  input inc:int;
  state s:int = 1;
  reaction(dbl) {= self->s *= 2; =}
  reaction(inc) {= self->s += inc; =}
}
reactor Relay {
  input r:int;    output out:int;
}

reaction(r) -> out {=
  lf_set(out, r->value); =}
}
reactor Y {
  output dbl:int;  output inc:int;
  reaction(startup) -> dbl, inc {=
    lf_set(dbl, 1); lf_set(inc, 1); =}
}

main reactor {
  x = new X();
  r = new Relay();
  y = new Y();
  y.dbl -> r.r;
  r.out -> x.dbl;
  y.inc -> x.inc;
}

```

Fig. 1. LF code of Relay.

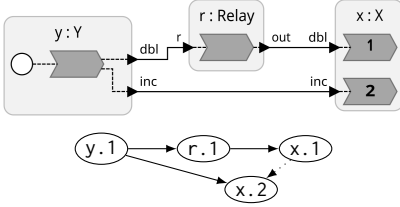


Fig. 2. Architecture and APG of Relay.

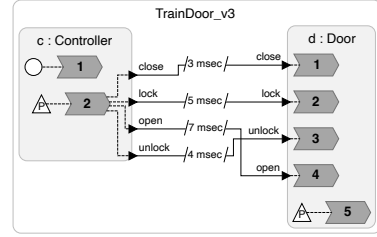


Fig. 3. Architecture of Train Door.

Example 2. Figures 3 and 4 show the LF model of a train door controller in [22]. The Controller reactor *c* has a Boolean physical action (marked 'P') external, modeling a driver requesting either to close-and-lock ((external->value == true)) or unlock-and-open the door ((external->value == false)); the former produces outputs on close and lock, the latter on open and unlock. All connections are *delayed*: the delay to lock is greater than to close, to close the door before locking it, and, similarly, to unlock before opening. The variables locked and isOpen in the Door reactor *d* model the state of the door. The Door has a physical action extOpen, modeling a passenger pushing an “Open Door” button, which results in the door being open(ed) (isOpen set to true), *unless* the door is locked. □

```

reactor Controller {
  output lock: bool
  output open: bool
  output unlock: bool
  output close: bool
  physical action external: bool
  reaction(startup) {=
    /* initialize system */ =}
  reaction(external) -> close,
    lock, open, unlock {=
    if (external->value == true)
    { lf_set(close, true);
      lf_set(lock, true); }
    else { lf_set(open, true);
      lf_set(unlock, true); } =}
}

reactor Door {
  input lock: bool
  input unlock: bool
  input open: bool
  input close: bool
  state locked: bool = false
  state isOpen: bool = false
  physical action extOpen: bool
  reaction(close) {=
    self->isOpen = false; =}
  reaction(lock) {=
    if (self->isOpen == false)
    self->locked = true; =}
  reaction(unlock) {=
    self->locked = false; =}
  reaction(open) {=
    if (self->locked == false)
    self->isOpen = true; =}
  reaction(extOpen) {=
    if (self->locked == false)
    self->isOpen = true; =}
}

main reactor {
  c = new Controller()
  d = new Door()
  c.lock -> d.lock after 5 msec
  c.unlock -> d.unlock after 4
    msec
  c.open -> d.open after 7 msec
  c.close -> d.close after 3
    msec
}

```

Fig. 4. LF code of Train Door.

Rewriting Logic and Maude. Rewriting logic [15] is a computational logic where data types are defined by an algebraic equational specification (Σ, E) —where the *signature* Σ declares *sorts*, *subsorts*, and *function symbols*, and E is a set of (possibly conditional) equations—and local state changes are modeled by (possibly conditional) labeled rewrite rules $[l] : t \longrightarrow t' \text{ if } \text{cond}$.

Maude [6] is a specification language and formal analysis tool for rewriting logic. A function f is declared `op  $f : s_1 \dots s_n \rightarrow s$` . Equations and rewrite rules are introduced with, respectively, keywords `eq`, or `ceq` for conditional equations, and `rl` and `crl`. They are universally quantified by the mathematical variables appearing in them,⁵ which may also be introduced on the fly and have form `var:sort`. An equation $f(t_1, \dots, t_n) = t$ marked `owise` (“otherwise”) can be applied to $f(\dots)$ only if no other equation with left-hand side $f(u_1, \dots, u_n)$ can be applied.

A declaration `class  $C \mid att_1 : s_1, \dots, att_n : s_n$` defines a *class* C with attributes att_1 to att_n of sorts s_1 to s_n . An *object* O of class C is represented as a term $<O : C \mid att_1 : val_1, \dots, att_n : val_n>$, where val_i is the value of att_i . A state is a term of sort *Configuration*, representing a *multiset* of objects and messages. System behavior is axiomatized by rewrite rules that specify transitions; e.g.,

```
rl [1] : <O : C | a1 : Y,    a2 : O2 >  <O2 : C2 | b : X >
      => <O : C | a1 : Y+X, a2 : O2 >  <O2 : C2 | b : Ø > .
```

defines a set of transitions in which an object O of class C adds the value of the attribute b (which is set to $\emptyset$) of the object $O2$ to its attribute $a1$. Attributes whose values do not change and do not affect the next state need not be mentioned.

Maude’s command `rew [n] init` simulates up to n steps of a behavior from the initial state *init*. The command `search [n] init =>* pattern [such that cond]` searches for at most n states reachable from *init* that match the state pattern *pattern* such that the (optional) condition *cond* holds. The command `red modelCheck(init,  $\phi$ )` checks whether the LTL formula ϕ is satisfied by the initial state *init*. Atomic propositions in ϕ are user-defined terms of sort *Prop*. The function `op  $_|\_ = \_ : \text{State Prop} \rightarrow \text{Bool}$` specifies which states satisfy a given proposition. LTL formulas are built from propositions, Boolean connectives, such as $\sim$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), and $\rightarrow$ (implication), and the temporal operators `[]` (“always”), `<>` (“eventually”), and `U` (“until”).

Real-time systems can be modeled as *real-time rewrite theories* [19], where ordinary rewrite rules model instantaneous change, and *tick* rewrite rules `crl [tick] : {t} => {t'} in time  $\tau$  if cond` model time advance, where the *whole state* has the form $\{u\}$. Time-specific analysis command, such as *timed* (or “metric”) CTL model checking [8], are provided by the Real-Time Maude tool [18].

3 Rewriting Logic Semantics for Lingua Franca

This section presents our executable rewriting logic semantics of a subset of LF. Section 3.1 explains how an LF program is represented as a Maude term, and Sections 3.2 and 3.3 describe how LF programs are executed in Maude.

⁵ We do not show declarations of variables, which are written in all capital letters.

Like other verification approaches for LF [9,22,13], we focus on logical time and do not treat physical time. In contrast to [9,22], where physical actions are either *unsupported* [9, p. 6] or translated into *fully deterministic* behaviors [22], we support a fairly general nondeterministic model of physical actions: each physical action has a “period” and a finite range of possible “values.” We then periodically choose nondeterministically whether or not the physical action takes place; if it takes place, its value is chosen nondeterministically.

In contrast to [13], we now also support multiple triggers of reactions, which makes it significantly harder to formalize the semantics of LF.

The reaction code is given in a target language (C, Rust, etc.). As explained in Section 6, we have extended the LF compiler to *automatically* transform reaction code in the fragment of C targeted by Lin et al. [9] into Maude.

3.1 Representing LF Programs in Maude

Extending [13], we formalize the semantics of LF in an object-oriented style. An LF program is modeled as a term where each reactor r is modeled as an object

```
< r : Reactor | inports : inPorts,    outports : outPorts, state : vars,
               reactions : reactions, timers : timers,    actions : actions >
```

$inPorts$ and $outPorts$ are sets of port objects $\langle name : Port \mid value : [v] \rangle$, $vars$ is a term $x_1 \mid \rightarrow [v_1] ; \dots ; x_n \mid \rightarrow [v_n]$, where v_i is the value of state variable x_i . $timers$ and $actions$ are sets of, respectively, Timer and PhysicalAction and LogicalAction objects. $reactions$ is a list of reactions, with each LF reaction $reaction(triggers) \rightarrow effects \{= code =\}$ modeled by a term $reaction \text{ when } triggers \rightarrow effects \text{ do } \{code\}$ (see [20,13]). We omit ‘ $\rightarrow effects$ ’ if no effects are declared.

An immediate connection is modeled as a term $(r : outPort \rightarrow r' : inPort)$ and a delayed connection as a term $(r : outPort \text{ -- } delay \rightarrow r' : inPort)$.

Example 3. The term `init` representing the initial state of the LF program in Example 2 is the following term (with omitted parts shown as ‘...’):

```
< c : Reactor | inports : none, timers : none, state : empty,
  outports : < lock : Port | value : [false] > < unlock : Port | value : [false] >
             < open : Port | value : [false] > < close : Port | value : [false] >,
  actions : < startup : LogicalAction | minDelay : 0, minSpacing : 0,
             minSpacing : 0, policy : defer, payload : [false] >
             < external : PhysicalAction | ..., payload : [false] >,
  reactions : (reaction when startup do {skip})
              (reaction when external --> close ; lock ; open ; unlock do {if ... fi}) >
< d : Reactor | outports : none, timers : none, state : (locked |> [false]) ; (isOpen |> [false]),
  inports : < lock : Port | value : [false] > < unlock : Port | value : [false] > ...,
  actions : < extOpen : PhysicalAction | ... >,
  reactions : (reaction when lock do {if isOpen == [false] then locked := [true] fi}) ... >
(c : lock -- 5 --> d : lock) (c : unlock -- 4 --> d : unlock) (c : open -- 7 --> d : open) ...   □
```

Defining the data types of these terms is straightforward (see [20] for details). For example, the class `Reactor` is defined as:

```
class Reactor | inports : Configuration,  outports : Configuration,
                state : ReactorState,      reactions : ReactionList,
                timers : Configuration,    actions : Configuration .
```

Extending [13], the states of our *executions* have the form

```
{< queue : EventQueue | queue : eventQueue >
 < r1 : Reactor | ... > ... < rn : Reactor | ... > ...
 < env : Environment | physicalActions : physActObjects >
 < rxns : Invoked | reactions : reactionsExecutedInLastStep >}
```

The queue object holds the (tag-ordered) event queue, with entries of the form: $events_1$ at $tag(t_1, n_1) :: events_1$ at $tag(t_2, n_2) :: \dots$, where $events_i$ is a set of events, t_i is the time *remaining* until $events_i$ should trigger reactions; and n_i is the microstep index. We do *not* store the actual time values of the tags, since they may grow unbounded—making the reachable state space infinite—even when the reachable “untagged state space” is finite.

The $< r_i : \text{Reactor} \mid \dots >$ objects model the reactors as explained in Section 3.1, and contain the current values of the reactor’s state variables.

The env object maintains data about physical actions, in objects

```
< r.a : PhysAct | leftOfPeriod : t,                period : p,
                  possibleValues : rangeOfValues,  timeNonDet : b >
```

r is the reactor, a is the action name, t is the time remaining until the action may happen next, p is its “period,” $rangeOfValues$ is the finite set of values the action can have, and b is a flag denoting whether the physical action is time-nondeterministic (i.e., does *not* have to take place each period). See [20,13].

Finally, unlike in [13], we now also store the set of reactions executed in the most recent step as the value *reactionsExecutedInLastStep*. This is essential for analyzing dataflow and “mixed” properties that cannot be handled in [13].

3.2 Defining the Dynamic Behaviors

We define the dynamic behaviors using three rewrite rules, in addition to those for physical actions: (1) a rule **step** defines the execution of one (big) step; and (2) two **tick** rules advance time until the first event(s) in the event queue are ready to be executed, or until a physical action timer expires.

The following rewrite rule performs a “big” (i.e., synchronous) step in the system: it executes all reactions triggered at the current logical time by the EVENTS at the *head* of the event queue, which have a *remaining* tag $tag(0, N)$:

```
cr1 [step] :
  {< E : Environment | physicalActions : CONF >  REACTORS-AND-CONNECTIONS
   < EQ : EventQueue | queue : (EVENTS at tag(0,N)) :: Q > < RXNS : Invoked | >}
=> {< E : Environment | physicalActions : CONF >  NEW-NETWORK
   < EQ : EventQueue | queue : NEW-Q > < RXNS : Invoked | reactions : INVOKED >}
if  smallestTimer(CONF) > 0
  /\  networkQueueRxns(NEW-NETWORK, NEW-Q, INVOKED) :=
      executeStep(EVENTS, REACTORS-AND-CONNECTIONS, Q) .
```

Executing the reactions triggered by EVENTS may trigger other reactions at the same logical time. All such reactions are executed by the *function* `executeStep`, which takes as arguments these EVENTS, the current state of the reactors and the connections (REACTORS-AND-CONNECTIONS), and the *remaining* event queue Q. It returns `networkQueueRxns(NEW-NETWORK, NEW-Q, INVOKED)`, where NEW-NETWORK is the network with the updated values of the state variables, NEW-Q is the event queue obtained by adding to Q the future events generated by the executed reactions, and INVOKED is the reactions executed in this “big” step.

The tick rewrite rule advances time until the remaining delay of the events at the head of the queue becomes 0, or until the next physical action expires:

```

cr1 [tick] :
  {< E : Environment | physicalActions : CONF > REACTORS-AND-CONNECTIONS
   < EQ : EventQueue | queue : (EVENTS at tag(T1,N)) :: Q > < RXNS : Invoked | >}
=> {< E : Environment | physicalActions : decreaseTimers(CONF, T) >
    REACTORS-AND-CONNECTIONS
    < EQ : EventQueue | queue : (EVENTS at tag(T1 minus T,N)) :: decreaseTags(Q,T) >
    < RXNS : Invoked | reactions : none >} in time T
if T := min(T1, smallestTimer(CONF)) /\ T > 0 .

```

All physical action timers decrease by the elapsed time T (`decreaseTimers(CONF, T)`) and all “remaining time” values in the tags in the event queue decrease by T (`decreaseTags(Q, T)`). A similar tick rule applies when the queue is empty.

3.3 Executing a Single “Big” Step

The function `executeStep` executes a single step; i.e., it executes all reactions triggered by the events at the head of the event queue, *and* those subsequently triggered at the same logical time by executing these reactions, and so on. We must also take various constraints into account. For example, in Example 1, reaction **2** in x cannot execute before reaction **1** has executed, *if* reaction **1** executes at this logical time. Whether or not a reaction execution produces output (and, if so, what) depends on the values of its state variables and inputs.

We *dynamically* maintain a *runtime acyclic precedence graph* (R-APG) while executing a big step. Given a set of events EVENTS, we construct an R-APG whose nodes are all the reactions that *possibly could* execute at the same logical time as those triggered by EVENTS. Each node is marked with one of four statuses: *executed* (the reaction has been executed in this step), *absent* (the reaction will not be triggered at this logical time), *unknown* (it is not yet known whether the reaction will be triggered at this logical time), or *present* (at least one of the reaction’s triggers is/will be present, but the reaction has not yet executed). Updating the R-APG and executing reactions in a big step must go hand in hand.

General Scheme. The function `executeStep` starts by building the initial R-APG for EVENTS, and invokes the main `executeStep` function maintaining the R-APG:

```

eq executeStep(EVENTS, NETWORK, Q)
  = executeStep(generateAPG(EVENTS, NETWORK),
    addEventsToPorts(EVENTS, NETWORK), Q, none) .

```

We define `executeStep` as a “fixed-point” function that executes *some* reaction r marked present in $rApp$, if all its predecessors are marked executed or absent:

```
eq executeStep( $rApp$ ,  $network$ ,  $queue$ ,  $reactionsInvoked$ )
= executeStep( $newRApp$ ,  $newNetwork$ ,  $newQueue$ ,  $newRxInvoked$ )
```

After executing a reaction r , we determine its outputs (if any) and update $rApp$ accordingly. For example, if executing r reveals that all triggers of r' will be absent, r' is marked absent, and the information that r' will not produce any outputs in this step must be propagated throughout the R-APG; this again could lead to other nodes going from unknown to absent, and so on.

Executing a reaction r may schedule future actions and/or output values to ports that are sources of *delayed* connections. Such future events must be added to the updated event queue $newQueue$. Executing r may also change its state variables, producing a new network $newNetwork$. We also add the executed reaction r to the set of reactions invoked in this step ($newRxInvoked$).

When no more reactions can be invoked, we return the new network (state), the new updated event queue, and the set of reactions invoked in this step:

```
eq executeStep(GRAPH, NETWORK, Q, INVOKED)
= networkQueueRxns(NETWORK, Q, INVOKED) [owise] .
```

It is worth remarking that the main equation executes *some* reaction r that is enabled. The key thing is that the semantics of Lingua Franca is deterministic, so that the order of execution of the reactions in a big step does not matter, as long as it respects the constraints represented by the APG.

This scheme can be seen as a *general scheme* for executing a set of “reactions” subject to dynamically changing “constraints”/“strategies” on those executions.

Main Equation. A node in the R-APG is an object of the following class:

```
class APGNode | triggers : TriggerStatus, pre : ReactionIdSet,
               succ : ReactionIdSet,      status : ExecutionStatus .
```

For *each* trigger of the reaction, the attribute `triggers` stores whether it is unknown, present, or absent. The attributes `pre` and `succ` denote the predecessor and successor nodes in the R-APG. The above main equation is then defined as:

```
ceq executeStep(
  < (REACTOR . N) : APGNode | triggers : TRIGGERS-STATUS, status : present, pre : PRE > GRAPH,
  NETWORK
  < REACTOR : Reactor |
    reactions : REACTIONS1 (reaction N when TRIGGERS --> OS do {BODY}) REACTIONS2 >,
  Q, INVOKED)
= executeStep(< (REACTOR . N) : APGNode | status : executed >
  updateGraph(GRAPH, REACTOR, OS, NETWORK < REACTOR : Reactor | >, OUTPUTS),
  propagateImmediateOutputs(OUTPUTS, NETWORK OBJECT),
  scheduleDelayedInputs(OUTPUTS, NETWORK, RESULT-Q), INVOKED ; (REACTOR . N))
if presetOK(PRE, GRAPH) /\ allTriggersDecided(TRIGGERS-STATUS)
/\ result(OBJECT, OUTPUTS, RESULT-Q) := executeReaction(< REACTOR : Reactor | >, N, TRIGGERS-STATUS, Q) .
```

In this equation, the reaction $(REACTOR . N)$ has status present in the R-APG, each of its predecessor nodes/reactions in the R-APG is either executed or absent

(presetOK(PRE, GRAPH)), and the status of each of its triggers has been decided (since all inputs must be consumed/read when a reaction is executed).

The function `executeReaction` executes the reaction (`REACTOR.N`), and returns the new state `OBJECT` of the reactor `REACTOR`, the outputs `OUTPUTS` generated by executing (`REACTOR.N`), and the event queue resulting from inserting the events scheduled in the execution of the reaction (`RESULT-Q`). The outputs `OUTPUTS` may be connected to *immediate* or *delayed* connections. The latter must be added to the event queue (`scheduleDelayedInputs(OUTPUTS, NETWORK, RESULT-Q)`), the former must be added to the corresponding input ports since they will trigger executions in this step (`propagateImmediateOutputs(OUTPUTS, NETWORK OBJECT)`). The R-APG must also be updated by the results of executing the reaction (`updateGraph(...)`).

Defining the functions `propagateImmediateOutputs`, `scheduleDelayedInputs`, and `executeReaction` is fairly straightforward. The function `updateGraph` is more subtle, and also uses a “fixed-point” style definition (see the Maude code for details).

4 Formal Analysis of LF Programs in Maude

Our formalization of the intended logical-time semantics of LF means that LF programs can be analyzed using Maude and Real-Time Maude. Existing formal analysis methods for LF are:

1. Sirjani et al. [22] use state-based reachability analysis.
2. Lin et al. [9] provide SMT-based *bounded model checking* for the restricted fragment of *safety MTL* where each temporal operator is annotated with a *finite* interval; e.g., $\Box_{[5,8]}(a \longrightarrow \Diamond_{[3,6]}b)$. Given a formula ϕ and model M , they compute a bound $\mathcal{CT}_{M,\phi}$, so that bounded model checking up to $\mathcal{CT}_{M,\phi}$ steps solves the model checking problem. Lin et al. [9] support three kinds of properties: (i) *state properties* consider the values of reactor variables, ports, and actions; (ii) *dataflow properties* are properties about *reactor invocations*; and (iii) *mixed properties* include both state and dataflow properties.
3. In [13] we provide unbounded and time-bounded reachability analysis and *untimed* LTL model checking, but only of *state properties*.

In this paper we provide more than all the analysis methods above, including:

- unbounded, step-bounded, and time-bounded reachability analysis for state, dataflow, and mixed properties;
- unbounded and time-bounded LTL model checking of all three kinds of properties; and
- *timed* CTL model checking, using Real-Time Maude, of all such properties.

Relation to Lin et al.’s Completeness Threshold. Lin et al. [9] use SMT solving to perform *bounded* model checking. They compute a *completeness threshold* $\mathcal{CT}_{M,\phi}$ so that ϕ holds in all behaviors if and only if ϕ holds in all behaviors of length $\mathcal{CT}_{M,\phi}$. The reason is that Lin et al. use a fragment of safety MTL where it is sufficient to consider behaviors up to time $h(\phi)$, which is less than or equal

to the sum of the upper bounds of the intervals of the temporal operators in ϕ . For example, for the formula $\Box_{[5,8]}(a \longrightarrow \Diamond_{[3,6]}b)$ it is enough to consider all behaviors up to time 14. Since Lin et al. consider an “event-based” semantics, and since they do step-bounded instead of time-bounded analysis, they must compute a bound on how many reactions could be invoked within time $h(\phi)$.

Since we support time-bounded analysis, we can model check the formula ϕ directly by performing time-bounded analysis with time bound $h(\phi)$.

Initial States. The initial state `initSystem` of our executions is represented as follows, where `init` is the initial state of an LF program (e.g., see Example 3):

```
eq initSystem =
{< env : Environment | physicalActions : physActs >
  addReactionIndices(init)
  < queue : EventQueue | queue : addInitialTimers(init, addStartup(...,empty)) >
  < rxns : Invoked | reactions : none >} .
```

where *physActs* contains a `PhysicalAction` object for each physical action in `init`; the function `addReactionIndices(init)` adds the “reaction number” to each reaction in `init`; the function `addInitialTimers` adds the initial timer events to the event queue; and the function `addStartup` adds desired actions that should trigger (at) the start of the system.

Example 4. The initial state `initSystem` for the train door system equals

```
{< env : Environment | physicalActions :
  < (c.external) : PhysAct | leftOfPeriod : 0, period : 10, timeNonDet : false,
    possibleValues : ([true] : [false]) >
  < (d.extOpen) : PhysAct | leftOfPeriod : 0, period : 11, timeNonDet : true,
    possibleValues : ([true]) > >
  addReactionIndices(init)
  < queue : EventQueue | queue : empty > < rxns : Invoked | reactions : none >}  □
```

4.1 Analysis Methods

By using different versions of the tick rewrite rule, we can perform unbounded, time-bounded, and/or “step-bounded” simulation and reachability analysis, as well as unbounded and time-bounded (untimed) LTL model checking directly in Maude [2]. Furthermore, we can also subject LF models to unbounded and time-bounded *timed CTL model* checking using Real-Time Maude.

Unbounded and Time-bounded Analysis. The tick rules in Section 3.2 are suitable for simulation, e.g., using the command `rew [10] initSystem`. However, they add a “system clock,” whose value may grow beyond any bound, to the state, making the reachable “clocked” state space infinite.

For *unbounded* analysis we use the tick rules in the module `UNBOUNDED-ANALYSIS-DYNAMICS` that do *not* add a clock to the state, by removing the “**in time T**” part from each tick rule `cr1 [tick] : {t} => {u} in time T if cond.`

For *time-bounded* analysis, we use the module TIME-BOUNDED-DYNAMICS that replaces each tick rule in our original semantics with the corresponding rule

```
crl [tick] : {t} in time T2 => {u} in time T2 + T if T2 + T <= timeBound /\ cond .
```

We must then define the value of `timeBound`, and use the initial state `initSystem` in time 0. In this way, we can do time-bounded and unbounded analysis by just including different modules.

Reachability Analysis. We can define search patterns on the execution state in search commands, including values of state variables, the invoked reactions in a state, the content of the event queues, and so on. The following *unbounded* search command checks whether it is possible to reach a (bad) state in which the state variables `isOpen` and `locked` in the door reactor both have the value `true`:

```
Maude> search [1] initSystem =>* {REST < d : Reactor | ATTS, state :  
                                (locked |-> [true]) ; (isOpen |-> [true]) ; RS >} .
```

No solution.

The variables `REST`, `ATTS`, and `RS` capture, respectively, the other objects in the state, the other attributes in the object `d`, and the other variables in `d` (superfluous in this case). It is worth noting that the reachable state space, *with* event queues, is finite in this case, so that a search for an unreachable state terminates.

To illustrate time-bounded analysis and *mixed* properties, we perform a *time-bounded* search to check whether the state variable `locked` could be `false` after a step in which the second reaction (the one triggered by `lock`) in `d` was executed:

```
Maude> search [1] initSystem in time 0 =>* {REST  
                                < d : Reactor | state : (locked |-> [false]) ; RS, ATTS >  
                                < rxns : Invoked | reactions : (d. 2) ; RXNS >} in time T .
```

Solution 1 (state 414)

```
REST --> ...   ATTS --> ...   RXNS --> ...   RS --> isOpen |-> [true]   T --> 35
```

The output shows that this situation can happen at time 35: the door is open when the lock signal arrives. The command `show path 414` shows the whole behavior.

LTL Model Checking. We can use Maude's linear temporal logic (LTL) model checker to perform both unbounded and time-bounded LTL model checking. We define the following *generic* atomic propositions:

```
op _in_is_ : VarId ReactorId Value -> Prop [ctor] .  
op _isInQueue : Event -> Prop [ctor] .  
op _invoked : ReactionId -> Prop [ctor] .
```

```
eq {REST < R : Reactor | state : (VAR |-> VAL) ; RS >} |= VAR in R is VAL = true .
```

```
eq {REST < 0 : EventQueue |  
    queue : (EQ :: (E at TAG) :: EQ2) >} |= E isInQueue = true .
```

```
eq {REST < 0 : Invoked | reactions : (RXN ; RXNS) >} |= RXN invoked = true .
```

```
ceq {REST} in time T |= PROP = true if {REST} |= PROP .
```

var in *reactor* is *value* holds if the current value of the state variable *var* in the reactor *reactor* is *value*. *e* *isInQueue* holds if event *e* is in the event queue, and *reaction* invoked holds if *reaction* was invoked in the step leading to the state.

Example 5. We check the *mixed* property whether it will happen in all behaviors that the door is unlocked even though the second Door reaction was just invoked:

```
Maude> red modelCheck(initSystem, <> (locked in d is [false] /\
                                     (d . 2) invoked)) .
```

This gives a counterexample: the controller *could* always want to open the door.

The main question is under what circumstances a very “pushy” passenger can keep the doors open forever, from some time on, no matter what the driver does. The easiest way to analyze this property is to change the `timeNonDet` flag of the physical action `d . extOpen` to `false` and check whether this guarantees that the doors will eventually stay unlocked forever:

```
Maude> red modelCheck(initSystem, <> [] (locked in d is [false])) .
```

This command returns `true` when the passenger pushes the button *every two time units*; otherwise there is a counterexample to this property: the controller *can* avoid being stuck with an unlocked door forever if the user cannot push the button more often than every three time units. $\square$

Timed CTL Model Checking. We can also perform Real-Time Maude analyses on LF models. For example, we can use Real-Time Maude’s timed CTL model checker to check whether the door *always* will open *within seven time units* after the driver wants them opened ($\forall \square (driverPushOpen \rightarrow \forall \Diamond_{\leq 7} doorOpen)$):

```
Maude> (mc-tctl initSystem |= AG ((event(c, external, [false]) isInQueue)
                                   implies AF[<= than 7] (isOpen in d is [true])) .)
```

This returns `Property satisfied`, but does not hold for bounds smaller than seven.

5 Case Studies and Benchmarking

In [9], 22 LF programs, drawn from real-world applications and other benchmarks, are used to evaluate LF VERIFIER. We have successfully analyzed all the desired (or stronger) properties for all those programs. Due to lack of space, this section presents only one small example. For more details and additional applications, including ADAS and the train door, see our report [20]. All the models and their analyses are available at <https://doi.org/10.5281/zenodo.18152767>.

Example 6. LF VERIFIER verifies the following *mixed* property of an aircraft door controller (see [20]): if the `ramp` variable of the `Vision` reactor initially is 0, then *in time 0* it should always be the case that *if* the first reaction (`0` in LF VERIFIER) of `door` is invoked, then `door`’s `doorOpen` variable must be 1:

Table 1. LF VERIFIER vs. Maude (execution times in seconds).

Program	LF VERIFIER		Maude	
	Gen.	Solving	Time	# States
ADAS	21.07	1.72	0.008	184
AircraftDoor	6.57	0.02	0.107	2
Alarm	5.36	0.01	0.030	4
CoopSchedule	21.1	0.31	0.002	4
Elevator	325.14	1,038.02	0.005	6
Election	41.65	3.27	0.001	6
Election2	12.21	0.17	0.001	6
Factorial	21.46	13.15	0.001	33
Fibonacci	94.96	153.49	0.010	33
PingPong	13.42	0.14	0.002	22
Pipe	142.41	12.15	0.001	14

Program	LF VERIFIER		Maude	
	Gen.	Solving	Time	# States
ProcessMsg	13.23	0.74	0.001	18
ProcessSync	4.99	0.01	0.001	6
Railroad	158.14	4.33	0.003	8
Ring	15.83	0.07	0.001	16
RoadsideUnit	74.07	0.62	0.001	2
SafeSend	6.48	0.03	0.001	4
Subway	29.96	0.31	0.001	5
Thermostat	13.41	0.14	0.037	2
TrafficLight	-	-	0.001	420
TrainDoor	5.65	0.01	0.001	2
UnsafeSend	6.5	0.02	0.001	6

```
(AircraftDoor_vision_ramp == 0) ==>
(G[0 sec](AircraftDoor_door_reaction_0 ==> (AircraftDoor_door_doorOpen == 1)))
```

Since this formula only concerns time 0, it can be checked in many ways. The following command analyzes the property in timed CTL in around 0.1 seconds:

```
Maude> (mc-tctl initSystem |= (ramp in vision is [0]) implies
      AG[<= than 0] (((door . 1) invoked) implies (doorOpen in door is [1])) .)
```

```
rewrites: 28135 in 102ms cpu (107ms real)          Property satisfied      □
```

We compare the performance of our analysis methods with LF VERIFIER on the LF VERIFIER benchmarks. Due to technical issues, we did not re-run the experiments on the LF VERIFIER, but use the numbers given in [9], which were done on a laptop running macOS version 11.7 with a 2.3 GHz 8-Core Intel Core i9 and 16GB RAM. For a fair comparison, we ran Maude on a laptop running macOS version 10.15.7, with a 2.6GHz dual-core Intel Core i5 and 8GB RAM. Table 1 summarizes the experimental results. For LF VERIFIER, “Gen.” shows the time taken to generate SMT encodings (including translation to UCLID5), and “Solving” is the time spent on SMT solving. For Maude, “Time” is the total analysis time, and “# States” is the number of the (synchronous) states explored.

Our analyses drastically outperform those in [9]. Even when considering SMT solving times, Maude achieves much faster analysis than LF VERIFIER, especially for complex models such as Elevator. This is because we execute many reactions in one step, resulting in fewer states. Since LF is *deterministic*, it is enough to execute all reactions in a big step in *some* order consistent with the APG, whereas the event-based model checkers consider *all* such sequences of reaction executions, which can be exponential in the number of reactions executed.

6 The LF-MC Verification Tool

We have implemented a tool, LF-MC, by extending the LF compiler (1fc), to:

- (i) automatically generate a Maude model from an LF program annotated with

intuitive queries, (ii) automatically generate the Maude analysis commands for those queries, and (iii) invoke Maude to execute those Maude commands. LF-MC supports unbounded and time-bounded simulation, reachability analysis, and LTL model checking, and is available at <https://github.com/symbolicsafety/lingua-franca-maude>.

The LF-MC user defines her properties in an intuitive property specification language by decorating her LF program with annotations of the form

```
@maude(analysis="simulation", timeBound=t, rewrites=n)
@maude(analysis="reachability", timeBound=t, rewrites=n, goal="stateFormula")
@maude(analysis="ltl", timeBound=t, goal="LTLformula")
```

where *timeBound* and *rewrites* (bound on the number of rewrite steps) are optional.

A *stateFormula* is a Boolean combination (with ! or ~ for negation, /\ for conjunction, \/ for disjunction, etc.) of *state propositions*. An *LTLformula* extends state formulas to LTL formulas using the temporal operators [], <>, U, W and O.

State propositions include: *r . n* invoked (the *n*th reaction in reactor *r* was invoked in the big step leading to the current state); *event(r, trigger, val)* inQueue, where *val* is optional (the event queue contains an event *event(r, trigger, val)*); and *relational expressions* over reactor variables and input ports.

Relational expressions are built from Boolean or integer constants, *var* in *r* (the value of the state variable *var* in reactor *r*), and *port* in *r* (the value of the input port *port* in *r*), combined using arithmetic operators (such as + and *), and relational operators (=, !=, and, for integer expressions, >=, >, <=, and <).

The user should also annotate her LF program to specify how to simulate each physical action: its range of possible values, its period, and whether or not the physical action *must* happen (*timeNonDet=false*) at the end of each period:

```
@maudePhysAct(name="action", inReactor="reactor", vals="values", period=p,
               timeNonDet=b)
```

values are written as an interval, e.g., 2 .. 5, or as a comma-separated list of values, and *b* is a Boolean value. *timeNonDet* is optional, with default value *false*.

Example 7. We annotate the LF program of the train door in Example 2 with:

```
@maudePhysAct(name="external", inReactor="c", vals="true, false", period=10, timeNonDet=true)
@maudePhysAct(name="extOpen", inReactor="d", vals="true", period=11, timeNonDet=true)
@maude(analysis="simulation", rewrites=10)
@maude(analysis="reachability", goal="(locked in d == true) /\ (isOpen in d == true)")
@maude(analysis="reachability", timeBound=150, goal="(locked in d == false) /\ (d.2 invoked)")
@maude(analysis="ltl", goal="<> ((locked in d == false) /\ (d.2 invoked))")
```

The first two define that the physical actions *external* and *extOpen* *may* (*timeNonDet=true*) happen every 10 and 11 time units, respectively, and the other annotations specify simulation, reachability analysis, and LTL model checking commands corresponding to the properties in Section 4. When we run *lfc* with this LF file as parameter, all the analysis commands are executed on the generated Maude model, and counterexamples are shown in an understandable form:⁶

⁶ LF-MC assumes that the time bounds in annotations are in milliseconds, and transforms them to nanoseconds, which is the time unit used by the LF compiler.

```

linux> lfc lf-maude/examples/src/TrainDoor_v3.lf
...
rewrite [10] in SIMULATION-TRAINDOOR : initSystem timeBound INF .
result ClockedSystem: { ... } in time 22000000 timeBound INF
=====
search [1] in ANALYSIS-TRAINDOOR : initSystem timeBound INF =>* {C:Configuration} timeBound TI:TimeInf
such that {C:Configuration} |= (d.sv.locked in d == [true] /\ d.sv.isOpen in d == [true]) = true .

No solution.
=====
search [1] in ANALYSIS-TRAINDOOR : initSystem timeBound 150000000 =>* ...

Solution 1 (state 188)      C:Configuration --> ...
=====
reduce in MODELCHECKER-TRAINDOOR :
modelCheck(initSystem timeBound INF, <> ((d . 2) invoked /\ d.sv.locked in d == [false])) .
result ModelCheckResult: counterexample(...)
==['actionHappens']==>
{ invoked: none    queue: event(d, d.pa.extOpen, [true]) at tag(0, 1)
  c :[inports: empty state: empty]
  d :[inports: ...   state: (d.sv.locked |-> [false]) ; d.sv.isOpen |-> [false]]}
==['step']==>
{ invoked: d . 5    queue: empty
  c :[inports: empty state: empty]
  d :[inports: ...   state: (d.sv.locked |-> [false]) ; d.sv.isOpen |-> [true]]}
==['extraTickRuleForPhysActs']==> ...

```

7 Related Work

We have described the differences between our contribution and all other work we know of providing formal verification for LF [9,22,13] elsewhere in this paper. Recent work [4] by the Berkeley group extends [9] to allow different languages (e.g., C and Rust) to define the code of different reactions. This work is somewhat orthogonal to ours, since it does not focus defining the semantics of LF. In particular, they keep the “small-step” *event-based* semantics of LF Verifier (Section VII.B: “This yields a nondeterministic polyglot model [...] where each transition executes one reaction”). This work therefore does not solve the problem of formalizing the desired “big-step” logical-time semantics of LF.

Rossel et al. [21] formalize a small-step operational semantics for a subset of LF without physical actions in the Lean theorem prover, and prove determinism of the Reactor model, but do support analyzing single LF programs.

Deantoni et al. [7] propose an operational semantics for LF using Gemoc Studio. They use their implementation primarily for interactive debugging, but if the state space is finite, they can perform exhaustive simulation to generate all traces, which are amenable to model checking using the CADP model checker. Their semantics does not coincide with the desired synchronous semantics of LF.

Rewriting logic has been used to formalize the semantics and to provide formal analysis to a range of programming and modeling languages [5,16,17]. The closest to our work is the formalization of Ptolemy II discrete-event models in Maude [3], but not even that work (or any other we know of) execute actions in lockstep with maintaining and updating constraints like we do in this paper.

8 Concluding Remarks

Extending our prototype in [13], we have formalized in rewriting logic the intended logical-time semantics of a large fragment of the LINGUA FRANCA (LF) coordination language for cyber-physical systems, which is well known to be a challenging problem. What made this possible was the use of a computational logic (rewriting logic) with both *rewrite rules*, defining transitions, and *equations*, defining functions (e.g., to execute a single big step). While we in [13] do not support multiple action triggers, and therefore do not cover most LF models, we now capture a subset of LF that contains all 22 LF VERIFIER benchmarks and all examples in [9,22,13]. The work in [9,22] does not consider (nondeterministic) physical actions. In such a setting, all models are deterministic. In contrast, we support a quite general nondeterministic model of physical actions.

Our formalization of the LF semantics enables analyzing LF programs using Maude and Real-Time Maude. In contrast to [13], we support unbounded and time-bounded simulation, reachability analysis, LTL model checking, and timed CTL model checking, for state properties, dataflow properties, and mixed properties, and can therefore analyze all properties in all LF VERIFIER benchmarks.

The papers [9,22] provide “small-step” event-based semantics for LF, which (i) is not the intended semantics of LF, and (ii) while they model check only deterministic models, they analyze all interleavings of the small steps, leading to inefficient analyses. We capture the big-step semantics, and benchmarking shows the dramatic difference in performance: our analyses take 0.1 seconds or less, whereas LF VERIFIER analysis can take more than 20 minutes.

We have extended the LF compiler to automate the Maude analysis of an annotated LF program, and provide easily readable output, so that the LF expert can analyze her programs without knowing Maude.

Lifting our view above LF, our formalization seems to provide a *general technique* for executing a set of actions whose execution order must satisfy a dynamically changing “strategy” or “constraint” (like the runtime-APG for LF).

Acknowledgments. We thank Edward A. Lee for helpful discussions about Lingua Franca, and also thank the anonymous reviewers for valuable comments on an earlier version of this paper. This work was supported by NATO SPS program project G6133. Rukhaia was supported by the Shota Rustaveli National Science Foundation of Georgia under the project FR-22-4254. Bae has been partially supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant (No. RS-2024-00439856) and by the National Research Foundation of Korea (NRF) grant (No. RS-2021-NR060080), both funded by the Korea government (MSIT).

Data-Availability Statement. The Maude semantics of LF, case studies, benchmark examples, and scripts to reproduce the experimental results, together with the LF-MC tool, are available at <https://doi.org/10.5281/zenodo.18152767>.

References

1. Aceto, L., Cimini, M., Ingólfssdóttir, A., Reynisson, A.H., Sigurdarson, S.H., Sirjani, M.: Modelling and simulation of asynchronous real-time systems using Timed Rebeca. In: FOCLASA. EPTCS, vol. 58, pp. 1–19 (2011)
2. Bae, K., Olarte, C., Ölveczky, P.C.: Modeling and analyzing real-time systems in rewriting logic. In: Concurrent Programming, Open Systems and Formal Methods: Essays Dedicated to Gul Agha to Celebrate His Scientific Career. Lecture Notes in Computer Science, vol. 16120. Springer Nature Switzerland, Cham (2026)
3. Bae, K., Ölveczky, P.C., Feng, T.H., Lee, E.A., Tripakis, S.: Verifying hierarchical Ptolemy II discrete-event models using Real-Time Maude. *Science of Computer Programming* **77**(12) (2012)
4. Chen, P.W., Lin, S., Godbole, A., Singh, R., Polgreen, E., Lee, E., Seshia, S.: PolyVer: A compositional approach for polyglot system modeling and verification. In: Proceedings of the 25th Conference on Formal Methods in Computer-Aided Design – FMCAD 2025. TU Wien Academic Press (2025)
5. Chen, X., Roşu, G.: $\mathbb{K}$ —a semantic framework for programming languages and formal analysis. In: SETSS. pp. 122–158. Springer, Cham (2020)
6. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude – A High-Performance Logical Framework, LNCS, vol. 4350. Springer, Berlin, Heidelberg (2007)
7. Deantoni, J., Cambeiro, J., Bateni, S., Lin, S., Lohstroh, M.: Debugging and verification tools for Lingua Franca in Gemoc Studio. In: 2021 Forum on specification & Design Languages (FDL). pp. 01–08 (2021)
8. Lepri, D., Ábrahám, E., Ölveczky, P.C.: Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories. *Science of Computer Programming* **99**, 128–192 (2015)
9. Lin, S., Manerkar, Y.A., Lohstroh, M., Polgreen, E., Yu, S.J., Jerad, C., Lee, E.A., Seshia, S.A.: Towards Building Verifiable CPS using Lingua Franca. *ACM Transactions on Embedded Computing Systems* **22**(5s), 1–24 (2023)
10. <https://www.lf-lang.org/>, accessed October 16, 2025
11. Lohstroh, M.: Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems. Ph.D. thesis, University of California, Berkeley (2020)
12. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a Lingua Franca for Deterministic Concurrent Systems. *ACM Transactions on Embedded Computing Systems (TECS)* **20**(4), 1–27 (2021)
13. Marin, M., Ölveczky, P.C., Reja, M., Rukhaia, M., Bae, K.: Semantics and formal analysis of Lingua Franca CPS specifications in rewriting logic. In: Rebeca for Actor Analysis in Action – Essays Dedicated to Marjan Sirjani on the Occasion of Her 60th Birthday, LNCS, vol. 15560. Springer (2025)
14. Menard, C., et al.: High-performance deterministic concurrency using Lingua Franca. *ACM Transactions on Architecture and Code Optimization* **20**(4), 1–29 (2023)
15. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.* **96**(1) (1992)
16. Meseguer, J., Rosu, G.: The rewriting logic semantics project. *Theor. Comput. Sci.* **373**(3) (2007)
17. Ölveczky, P.C.: Semantics, simulation, and formal analysis of modeling languages for embedded systems in Real-Time Maude. In: Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000. Springer (2011)

18. Ölveczky, P.C.: Real-Time Maude and its applications. In: Proc. WRLA'14. LNCS, vol. 8663. Springer (2014)
19. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. *Theor. Comput. Sci.* **285**(2), 359–405 (2002)
20. Ölveczky, P.C., Reja, M., Rukhaia, M., Bae, K., Marin, M.: Efficient verification of Lingua Franca programs (longer report) (2025), available at <https://the-mrd.github.io/tacas-artifact>
21. Rossel, M., Lin, S., Lohstroh, M., Castrillón, J., Goens, A.: Provable determinism for software in cyber-physical systems. In: *Verified Software. Theories, Tools and Experiments (VSTTE 2023)*. LNCS, vol. 14095. Springer (2023)
22. Sirjani, M., Lee, E.A., Khamespanah, E.: Verification of cyberphysical systems. *Mathematics* **8**(7), 1068 (2020)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





A Case Study in Firmware Verification: Applying Formal Methods to Intel® TDX Module

Dirk Beyer¹, Po-Chun Chien¹, Bo-Yuan Huang²,
Nian-Ze Lee^{3,1}, and Thomas Lemberger¹

¹ LMU Munich, Munich, Germany

² Intel INT31, USA

³ National Taiwan University, Taipei, Taiwan

Abstract. Firmware underpins system security but remains challenging to verify due to hardware dependency, specialized coding idioms, and limited open-source examples. Manual verification approaches, while common in industry, are labor-intensive and difficult to scale. This paper presents a detailed case study on applying automatic formal methods for software to a security-critical firmware component in Intel® Trust Domain Extensions (TDX), known as TDX Module. In this study, we employ six state-of-the-art C-program analyzers on the production TDX Module firmware, leveraging techniques ranging from bounded model checking and symbolic execution to abstract interpretation. Our empirical evaluation identifies obstacles unique to firmware, highlights harness-design decisions essential for verifying industry-scale code bases, and demonstrates opportunities in advanced slicing for more scalable verification. Although the case study focuses on TDX Module, the findings are broadly applicable to large-scale, low-level programs and have already influenced the software-verification community, such as standardizing nondeterministic object initialization. All verification tasks and proof harnesses are publicly released to foster reproducible research and future tool development.

Keywords: Firmware verification, Software verification, Industrial verification methodology, Trusted execution environment, Intel TDX

1 Introduction

Firmware lies at the foundation of modern computing platforms, orchestrating the boot process, hardware initialization, and security enforcement. Operating at the lowest layer of the stack in privileged and opaque environments, any defect in firmware can compromise the trustworthiness of the entire system. Despite its critical role, firmware verification in practice remains largely manual, relying on code reviews and dynamic testing through handcrafted test suites or fuzzing. For instance, Microsoft [37] and Google [3] recently conducted security reviews of Intel® Trust Domain Extensions (TDX) [1], which provides a hardware-based

An appendix to this article is available on our supplementary webpage [11].

isolation mechanism to protect *data in use* at the virtual-machine (VM) level. Such expensive and laborious reviews highlight the pressing need for automated, rigorous formal reasoning for firmware.

1.1 Problem Statement: Applying Software Verification to Firmware

We aim to transform today’s largely manual firmware-verification process into a systematic, tool-supported workflow. Since firmware is mostly written in C [44], recent advances and standardization in verification techniques and tools for C programs [17] provide a strong basis for this effort. This case study examines *how to apply automatic software-verification techniques to firmware to efficiently and effectively check specification conformance and detect vulnerabilities*.

While many success stories on applying automatic C-program verifiers to industry-scale code bases have been reported [7, 16, 22, 23, 47], and several projects worked towards streamlining the transformation from code projects to verification tasks [4, 14, 29, 47, 51, 59, 65, 74, 76], their application to firmware remains scarce. We discuss the key challenges of firmware verification and opportunities offered by formal methods in the following:

Hardware Dependency. Arguably the most referenced challenge in analyzing firmware programs, hardware dependency breaks the levels of abstraction that automatic tools often depend on. Dynamic testing tools, for example, cannot be deployed until late in the development cycle after the hardware has been fabricated. Operational virtual prototypes and formal abstraction models help left-shift the process [35, 41, 42, 61, 64, 73], but they incur non-trivial engineering costs. Planning and assessing such costs is a key part of firmware verification to ensure that early modeling efforts align with the system’s design considerations.

Negative Space. Security-critical firmware, such as TDX Module, must be robust against adversarial inputs and unexpected scenarios. Beyond verifying correct behavior under expected conditions, the *positive space*, security assurance requires rigorous input validation and error handling in adversarial or unforeseen cases, known as the *negative space*. While the positive space is typically addressed through manually crafted test cases during development, the negative space is harder to cover due to the difficulty to enumerate and the sheer size of the space. Compared to fuzzing or dynamic testing, the symbolic nature of formal verification provides the exhaustive rigor demanded by security-critical firmware, motivating our investigation into its application to firmware analysis.

Stateful Analysis. The stateful nature of firmware adds another layer of complexity to its verification. The behavior of a firmware component may depend not only on its inputs but also on the firmware and hardware states. This state-dependent behavior remains an open challenge for automatic testing techniques, which often require user-defined transition models, a suite of predefined input sequences, or design-specific code instrumentation [5, 54]. In contrast, formal methods enable modular verification of individual procedures by symbolically initializing and constraining the state variables accessed by the procedure under verification. No additional setup sequences or user-defined transition models are required.

1.2 Intel Trust Domain Extensions (TDX)

Intel TDX [1] is Intel’s latest confidential computing technology [62, 68] that provides an isolation mechanism to protect *data in use* at the virtual-machine (VM) level, in addition to the process-level isolation provided by Intel SGX [28, 57]. It is a hardware-based trusted execution environment (TEE) [33, 45, 63, 67] that facilitates the deployment of hardware-isolated VMs, called *Trust Domains* (TDs), and provides memory and CPU-state confidentiality and integrity as well as address-translation integrity. Moreover, it isolates TDs from the host virtual-machine manager (VMM), hypervisor, and other non-TD software to further reduce the trusted computing base of a TD.

Intel TDX introduces several new architectural elements, including TDX Module, a security service module running in a new, most privileged, CPU execution mode called Secure Arbitration Mode (SEAM). TDX Module enables controlled interactions between the host VMM and TDs through a set of application binary interface (ABI)

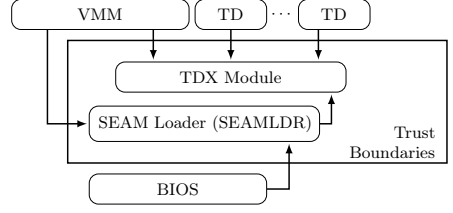


Fig. 1: Trust boundaries of TDX Module

functions. For example, host-side interface functions allow the host VMM to initialize TDX Module, configure keys, and create and manage TDs. Meanwhile, guest-side interface functions allow a TD to access platform metadata, create attestation-measurement reports, request host VMM services, etc. Besides supporting the core operations, more importantly, TDX Module is responsible for enforcing the security policies of the platform. Even more so, the ABI presents a prime attack surface as it receives untrusted inputs from outside of the trust boundaries, as shown in Fig. 1. This makes verifying TDX Module essential for the overall security assurance of Intel TDX.

The challenges outlined in Sect. 1.1, hardware dependency, negative space, and stateful analysis, are inherent in TDX Module: For example, hardware dependency can be seen in the use of the extended instruction set for measurement and remote attestation, which require appropriate modeling. Meanwhile, as TDX Module’s interface functions receive untrusted inputs from outside the trust boundary, this necessitates rigorous validation to cover the negative space. Nonetheless, the execution semantics of these functions are stateful, conditioned on the values of registers and control structures at the time of execution.

1.3 Overview of the Adopted Methodology

To enable the application of off-the-shelf software analyzers to industrial firmware, we adopt a common verification methodology [23] and tailor it to address the unique challenges of firmware. Our methodology comprises three key techniques, each targeting one of the challenges identified in Sect. 1.1.

To handle hardware dependency, we employ a manual CEGAR-like [24] approach for abstracting the underlying hardware: Firmware-hardware interactions (e.g., memory-mapped I/O and SEAM operations) are first coarsely overapproximated.

mated based on their signatures, returning nondeterministic values and havocking⁴ mutable arguments. These overapproximations are refined on demand only when they cause false alarms, minimizing modeling effort while maintaining soundness.

To enable effective negative-space verification, we develop systematic proof-harness engineering practices that reflect industrial verification workflows [23]. Each verification task is automatically extracted from unmodified production code using a lightweight proof harness that (i) sets up the execution context through lazy symbolic initialization, (ii) encodes preconditions capturing the negative-space scenario of interest, and (iii) specifies postconditions as verification targets. Harness components are compositional and allow for automatic task creation, enabling systematic management and reuse across verification tasks.

With symbolically initialized state variables, we define modular, type-based constraints that are reusable invariants over firmware data structures. Rather than exhaustively enumerating all possible system states, these constraints characterize valid states declaratively by type, enabling type-based initialization that instantiates and constrains only the state variables accessed by the function under verification. This compositional approach [23] provides modular verification of individual interface functions.

Together, these three techniques transform the firmware-verification problem into a form tractable for existing software-verification tools while maintaining the rigor required for security-critical systems.

1.4 Case Study: Insights from Intel TDX Module Verification

We validate this verification methodology through a detailed case study of Intel TDX Module, developing hardware abstractions, proof harnesses, and verification properties for 22 representative interface functions to enable the application of software-verification techniques implemented in off-the-shelf C analyzers. In total, 418 verification tasks are derived from the unmodified TDX Module code base. Each task is a single-file, self-contained C program, automatically extracted using HARNESSFORGE,⁵ a tool that performs syntactic program slicing on complex code bases and keeps only relevant components for the given target functions. The total human effort for this case study was approximately ten to twelve person-months.

Evaluation with six state-of-the-art tools (CBMC [25], ESBMC [32], KLEE [20], CPAchecker [15], MOPSA [46], and UAutomizer [39]) reveals fundamental insights across three dimensions. First, design decisions in proof-harness engineering critically impact effectiveness, with lazy initialization improving success rate and tool-specific havocking outperforming generic methods. These findings have motivated the adoption of a standardized mechanism for object havocking in SV-COMP 2026.⁶ Second, the capabilities of off-the-shelf software verifiers remain limited for firmware: Only 59 % of the tasks were solved by any of the evaluated tools. Carefully inspecting the unsolved tasks indicates that *property-guided slicing* (details in Sect. 6) presents a key direction for improvement by considering

⁴ In program analysis, “havoc” means assigning a nondeterministic value to a variable.

⁵ <https://gitlab.com/sosy-lab/software/harnessforge>

⁶ https://gitlab.com/sosy-lab/sv-comp/bench-defs/-/merge_requests/544

the semantics of the property under verification to slice the code base. Such optimization is particularly effective for negative-space verification of security-critical programs like TDX Module. Third, firmware-specific characteristics such as nested type punning and compiler memory-layout attributes break software verifiers’ frontends and hinder applicability. In our evaluation, half of the tools failed on all tasks largely due to these patterns.

Our study highlights the importance of viewing verification workloads as a combination of the target-under-verification and proof harness, emphasizing the need for type-based modularization and tool standardization to enhance verification engineering and drive technical innovations.

Contributions. In the following, we summarize the contributions of our paper:

1. We adapt an established methodology for code-level model checking and tailor it to firmware by integrating hardware abstractions and modular, type-based property and constraint development. This approach streamlines the application of formal methods to industry-scale firmware (details in [Sect. 3](#)).
2. We validate the adapted methodology on Intel TDX Module, focusing on negative-space verification of its interface functions. The study develops formal abstractions for architectural extensions relied upon by TDX, engineers proof harnesses that emulate industrial verification workloads, and derives type-based constraints from the official specification [1] (details in [Sect. 4](#)).
3. Using the developed infrastructure, we automatically extract 418 SV-COMP-compliant [17] single-file verification tasks from the production TDX Module code. Six state-of-the-art analyzers featuring representative techniques, including symbolic execution, bounded model checking, and abstract interpretation, are applied, which constitutes the first large-scale assessment of automatic formal software verification on commodity firmware (details in [Sect. 5](#)).
4. We analyze how firmware-specific characteristics, property types, and harness-design decisions affect verification performance. The study identifies persistent obstacles and distills practical insights to guide future industrial and academic verification efforts (details in [Sect. 6](#)).
5. The developed hardware abstractions, properties, proof harnesses, and the derived verification tasks are publicly released to support reproducible research and foster future development in this critical domain.

2 Related Work

Our study builds on previous work on formal verification of low-level programs, including device drivers, boot loaders, and embedded firmware [8, 42, 48, 50, 61, 64, 66]. The formal modeling of hardware abstraction is especially underpinned by the foundational studies in interfacing and reasoning about software-hardware interactions [35, 40, 41, 43, 56, 73]. We first review previous efforts in formalizing and verifying TEE architecture designs, and then discuss the related work on verifying the security monitor implementation in other TEE platforms.

High-Level TEE Formalization and Verification. The research community has been formalizing TEE systems at the design level to prove functional

correctness and key security properties [36, 55, 58, 69, 72, 75, 77]. On the commodity TEE side, Intel developed DVF, iPave, and Accordion that provide the languages and tools to formally model and verify Intel SGX [31, 34, 52]. Also, addressing VM-based TEE, recent work has formally modeled the attestation protocol of Intel TDX and proved the confidentiality of the shared secret and authentication of reports [70, 71].

This existing body of research in formalization and verification of hardware-assisted TEE is essential in ensuring a correct and secure design. However, the actual implementations of TEE can still have security bugs even if their high-level models have been formally verified. We emphasize the importance of directly verifying the actual implementation; for the same reason, our study considers properties that are low-level and implementation-oriented, in contrast to the higher-level properties discussed in the literature.

TEE Security Monitor Verification. Compared to design-level formalization, few works target implementation-level verification of TEEs. The Verification Infrastructure for Armv9-A (VIA) verifies an early prototype of the Realm Management Monitor (RMM) within the Arm Confidential Compute Architecture [53]. The approach is based on the interactive theorem prover Coq [2, 27], and thus, while highly expressive, requires significant manual efforts. In contrast, automatic tools CBMC and ESBMC have been applied to the reference implementation of RMM [30, 79], where the RMM implementation is automatically instrumented with pre- and postconditions derived from a machine-readable specification.

Our case study builds on learnings from these two success stories. While they rely on certain tool-specific customizations in their workflows, we focus on generating tool-agnostic verification tasks that comply with widely adopted conventions in the SV-COMP [17] and Test-Comp [9] communities. This allows leveraging the advances in these communities and unlocks verification techniques beyond bounded model checking.

3 Methodology for Automated Firmware Verification

We adapt the established methodology for code-level model checking that reflects industry best practices, similar to the approach developed at Amazon Web Services for verifying cloud software [23]. Given a function under verification, this methodology introduces a proof harness that initializes data structures used by the function symbolically, assumes preconditions over the data structures declaratively with type-based constraints, calls into the function, and asserts postconditions after the function returns. The following subsections detail key components we introduce to tailor this methodology for firmware verification.

3.1 Hardware Abstraction

Firmware interacts directly with hardware through low-level mechanisms such as inline assembly and memory-mapped I/O. Capturing these interactions, i.e., modeling the states and semantics of the operations, is essential for the complete verification of firmware. Industrial firmware code bases usually use dedicated wrap-

per functions to interface with hardware-dependent components. This modular design allows clean override of these functions with abstraction models.

Regarding inline assembly, uses of standard instructions usually have well-defined semantics and can be directly mapped to C implementations. For architectural extensions, we begin with a coarse overapproximation based on the signature of a wrapper function. The abstraction returns a nondeterministic value and updates all mutable inputs with nondeterministic values as well. We refine the abstraction only when it causes false alarms in verification results, similar to the CEGAR approach [24].

Firmware programs access reserved memory regions that map to hardware registers, data initialized by earlier stages of the boot process, etc. Efficiently handling these regions is crucial for firmware verification and often requires specialized tool support. For example, CBMC [25] provides a hook for memory-mapped I/O [26]. In our case study, we introduce mock variables to represent the states of the reserved memory regions accessed by TDX Module.

3.2 Proof-Harness Engineering

To enable effective negative-space and stateful verification of firmware, we develop a proof harness to create the verification context for each target function under verification. The proof harness contains a setup procedure that prepares the verification context by allocating and initializing the necessary state variables and input arguments. During this step, assumptions are given to relevant input arguments and state variables for capturing the negative space of the function and reflecting the stateful nature of firmware. On the other hand, we develop a corresponding teardown procedure that deallocates the state variables after the verification task is finished. Below we describe the crucial considerations for the setup and teardown procedures.

Allocation and Deallocation. While some state variables are defined as global variables in the code, others—such as control structures initialized by earlier booting stages—are opaque to the program. For these, we explicitly allocate memory in the setup procedure and deallocate during teardown.

To keep the proof harness lightweight and verifier-friendly, we allocate only the state variables accessible by the function under verification based on the specification. This *lazy* approach not only simplifies the verification context but also helps to identify potential mismatches between the implementation and the specification in terms of the accessible states.

Symbolic Variable Havocking. To cover the entire state space, the setup procedure symbolically initializes and havoc state variables and input arguments. Our proof harnesses support three styles for symbolically havocking variables with non-primitive, complex data structures.

1. *Memory havocking.* Each byte of the variable is assigned a nondeterministic value of type `unsigned char`.
2. *Object havocking.* Based on the type definition, boilerplate code is automatically synthesized to havoc the variable. Fields of primitive types are assigned

nondeterministic values of the corresponding type, while fields of non-primitive types are recursively traversed till primitive types are reached.

3. *Tool-specific havocing*. Besides the above two generic approaches, our proof harnesses can be configured to use tool-specific primitives for symbolic havocing, such as the method `__CPROVER_havoc_object()` in CBMC [25].

Note that if a non-primitive type includes a pointer field, assigning a nondeterministic value to the pointer does not yield a valid pointed-to object. Therefore, we need to allocate and initialize the pointed-to object and assign its address to the corresponding pointer field.

Hoare-Style Preconditions and Postconditions. After symbolically initializing the state and input variables, Hoare-style preconditions and postconditions specific to the property under verification are derived from the specification and encoded in the proof harness. While table-like data structures often need conditions over individual elements, which can naturally be expressed with universal quantification, we use loops to iterate over elements in the proof harnesses as off-the-shelf verifiers typically do not support quantifiers.

3.3 Type-Based Constraints

Specifying constraints over variables is one of the most labor-intensive aspects of developing a proof harness. For reusability and composability, we define these constraints in a type-based and declarative manner [23]. They fall into two main categories: (1) *value conditions*, such as requiring the state variable of a finite-state machine to belong to a predefined set or an address to lie within the physical-address range; and (2) *structural conditions*, such as ensuring no overlap between memory regions recorded in a metadata structure. These type-based constraints play a crucial role in defining inductive invariants of the system state, which are assumed in the setup procedure and checked in the teardown procedure for their inductiveness.

While type-based constraints capture a significant portion of the required conditions, some constraints cannot be expressed in a type-based manner. This is especially prominent for conditions that involve relationships between multiple variables. We specify such inter-variable relationships separately in the proof harnesses.

3.4 Automatic Verification-Task Generation

Fully automatic software verifiers for C programs typically assume self-contained, single-file verification tasks with all necessary data structures and function definitions explicitly included [17, 19]. This limits direct application of these verifiers to industrial code bases. Specifically, extracting the relevant components and assembling them into verification tasks for off-the-shelf tools is non-trivial and error-prone without systematic methodology.

We use the tool `HARNESSFORGE` to assemble a standalone verification task from the proof harnesses and the firmware code base under verification. `HARNESSFORGE` takes as input a YAML configuration file that defines a verification task with (1) a sequence of method calls and (2) any overrides of original project source code.

This enables automatic generation of verification tasks by specifying the sequence of calls to harness components and the hardware abstractions as overrides.

From this configuration, HARNESSFORGE computes the relevant project sources, applies the source-code overrides, and creates a verification entry that calls the specified methods in the given order. Moreover, HARNESSFORGE performs a dependency analysis based on the method call tree to remove all methods, global variables, and type definitions that are irrelevant to the verification task.

4 Design of Case Study on TDX Module

We apply the methodology described in Sect. 3 to TDX Module [1], a security-critical, open-source firmware component of the Intel TDX technology. We focus on TDX Module’s ABI functions, which receive untrusted inputs from callers outside the trust boundaries and thus warrant comprehensive negative-space verification. Our goal is to investigate how formal software verifiers can be leveraged with the adapted verification methodology to achieve high coverage and assurance for these security-critical interface functions.

This section describes the design of the case study, including the developed hardware abstractions, the selection of target interface functions, and the derivation of verification properties.

4.1 Abstraction of TDX Module’s Hardware Dependencies

The hardware dependencies of TDX Module fall into three categories:

1. *Executing standard x86 instructions.* TDX Module uses inline assembly to execute standard x86 instructions in scenarios such as reimplementing standard library functions (cf. Sect. 6), ensuring atomic operations, or handling wide bit-width data types that would otherwise be inefficient in C. While specific to the hardware platform, these instructions are normally design-agnostic.
2. *Architectural extensions.* As part of its core function, TDX Module manages the underlying hardware and leverages its computational capabilities by accessing model-specific registers and utilizing extended instruction sets. For example, to support measurement and attestation, it invokes the newly introduced instructions SEAMDB_REPORT and SEAMVERIFYREPORT to generate and verify reports, respectively. These interactions tend to be highly design-specific and require domain expertise for precise modeling.
3. *Accessing reserved or memory-mapped regions.* Data accesses in firmware are not always explicitly defined in its source code. For example, memory-mapped regions may not be declared by the firmware program, yet their accesses are valid and managed by the memory controller. In TDX Module, such “undefined” data accesses occur when accessing data initialized by SEAMLDR, the bootloader module in Intel TDX, which determines the physical addresses of TDX Module code and data based on the BIOS configuration and updates the corresponding dedicated registers.

The TDX Module code base uses dedicated wrapper functions to interface with hardware-dependent components, allowing clean override of these functions with

abstraction models. As described in Sect. 3.1, we translate standard x86 instructions to shadow C implementations directly. For the design-specific extensions leveraged by TDX Module, we overapproximate their behaviors based on the signatures of the wrapper functions by returning nondeterministic values of the expected return types and havocking all mutable inputs. For reserved or memory-mapped regions, mock variables are introduced to represent their states; constraints on these variables are added as needed to reflect the verification context.

In this case study, we observe that most of the first-attempt signature-based abstractions are *sufficient* for proving our selected properties and do not require further refinements.

4.2 Selection of Target Interface Functions

TDX Module’s ABI includes 77 host-side and 31 guest-side interface functions. All functions are security-critical and warrant comprehensive verification; in this case study, we focus on a representative subset to investigate the effectiveness of the adapted verification methodology. The selection is guided by the following criteria and objectives:

- **Interface-function class:** TDX Module ABI functions are grouped into classes based on their functionalities, such as physical-memory management, migration supports, and measurement and attestation. Each class performs distinct operations, managing different parts of the system state and following a unique program flow. To ensure broad coverage, our case study includes interface functions from each class.
- **Module and TD lifecycle:** TDX Module enforces platform security policies by ensuring that the lifecycles of both the module and TDs comply with the defined protocols. These protocol state machines guide and track these lifecycles through stages such as initialization, operation, and shutdown. Different sets of interface functions are permitted at different lifecycle stages. Our case study covers functions permitted in each stage and includes all functions that can trigger lifecycle-stage transitions.

Using the above criteria, we include 16 host-side and 6 guest-side interface functions into the case study (cf. Table 2).

4.3 Selection and Derivation of Verification Properties

Security assurance is a multi-faceted effort. For complex systems like Intel TDX, it requires a combination of verification techniques applied at various abstraction levels and targeting different classes of properties. This case study focuses on a subset of properties that (1) can be verified modularly at the individual interface-function level and (2) are practically tractable using off-the-shelf software analyzers. This results in safety properties, which address *functional correctness* and *state integrity*, as well as reachability properties, which are introduced to cover different execution paths within a function.

For additional developer-defined properties, such as debug assertions and sanity checks embedded in the TDX Module source code, we added preprocessor

Table 1: Excerpted specification of the interface function `TDH.MNG.KEY.CONFIG`, showing its input and output operands, access information on state operands, and possible completion-status codes

Operand	Description
Input	
RAX	SEAMCALL instruction leaf number and version
RCX	The physical address of a TDR page (HKID bits must be 0.)
Output	
RAX	SEAMCALL instruction return code
Other registers	Unmodified
State Information	
Trust Domain Root (TDR) page	Explicitly accessed, Read/Write permission, 4KB alignment
Key Encryption Tables (KETs)	Implicitly accessed
Completion-Status Codes (Excerpted)	
<code>TDX_OPERAND_INVALID</code>	An input operand of the interface function is invalid.
<code>TDX_LIFECYCLE_STATE_INCORRECT</code>	System is in an incorrect lifecycle state for this interface function.
<code>TDX_KEY_GENERATION_FAILED</code>	CPU failed to generate a random key.

directives to distinguish them from the above properties. They are omitted from the evaluation in [Sect. 5](#) due to space constraints. Other classes of properties, e.g., liveness, noninterference, and information flow, which arise in sequences of interface-function invocations or concurrent interaction between multiple TDs, are important for comprehensive guarantees of data confidentiality and integrity, but left for future work as they do not enjoy the same level of tool readiness.

We derive the properties based on the TDX Module ABI specification [1], applying best-effort interpretation in cases of ambiguity or missing details. As an example, [Table 1](#) provides an excerpt of the specification for the interface function `TDH.MNG.KEY.CONFIG`, which is used to configure a TD’s private key. For each interface function, the specification defines the input and output operands, along with their expected conditions and values. In addition to register-based input and output operands, it documents the firmware and hardware states that the interface function may access or modify. It also lists the possible completion-status codes, accompanied by descriptions of the corresponding failure conditions.

Functional-Correctness Properties. TDX Module ABI functions receive untrusted inputs from callers outside the trust boundaries. As a result, they perform *precondition checks* and *error handling* before executing their core logic. In this case study, we derive functional properties that capture such behavior, with a focus on the negative space, i.e., cases where preconditions are violated. Specifically, the properties verify that the interface functions detect invalid inputs or states and respond accordingly, such as clearing sensitive data and returning the designated error code. For each interface function, we enumerate detailed preconditions based on the available information in the specification [1]. An example proof harness for invalid input handling in `TDH.MNG.KEY.CONFIG` is shown in [Appendix A](#) [11].

State-Integrity Properties. As shown in Table 1, the specification defines the set of registers and states that an interface function may access or modify. We include properties asserting the integrity of registers that should not be modified by a given interface function. Integrity properties for states with complex data structures are excluded, as no available tools were found capable of handling them.

Reachability Properties. Based on the specification, we derive reachability properties to check that the target interface function can indeed complete with different status codes. These codes represent various precondition-check failures or convey distinct outcomes upon successful completion. The reachability properties guide the analysis of different execution paths within the interface function.

Besides the above properties derived from the specification, we also systematically generate *cover tasks* by inserting reachability labels at various control points within a task. Specifically, we insert labels at the completion of (1) the preconditions, (2) the target function call, and (3) the teardown procedure. Cover tasks provide an approximate measure of how effectively a software analyzer explores the verification context. Inserting them is a common industrial practice since fully converged proofs (i.e., properties under verification proved or disproved) are not always attainable, and cover tasks provide evidence of analysis progress, analogous to coverage metrics in testing.

5 Results and Effects of Harness-Engineering Decisions

This section presents the outcomes of applying our verification methodology to TDX Module and an analysis of how different initialization and havocking strategies in the proof-harness design influenced the effectiveness of the evaluated software analyzers. We discuss the key challenges encountered and the underlying reasons for the limited effectiveness of current verification tools.

5.1 Verification Setup

Automatic Analyzers. We leveraged six state-of-the-art software analyzers for C programs, encompassing different classes of verification techniques. The tools include the industrial-strength bounded model checkers CBMC (version 6.6.0) [25, 49] and ESBMC (version 7.7.0) [32, 78], the symbolic-execution engine KLEE (version 3.1) [20, 21], and three top-ranking participants from SV-COMP 2025 [17]: CPAchecker (version 4.0) [6, 15] and MOPSA (version 1.0) [46, 60], the first- and second-place winner in category *SoftwareSystems*, respectively; and UAutomizer (version 0.3.0-d790fccc) [38, 39], the overall competition winner.

Among these tools, CBMC, ESBMC, and KLEE offer built-in primitives for symbolic initialization of complex data structures, which we leveraged for tool-specific havocking. We executed each tool with the configuration used in its most recent competition participation. For CBMC and KLEE, we adapted their latest competition wrapper scripts to ensure compatibility with their current releases.

Verification Tasks. We derived the verification tasks from version 1.5.05 of TDX Module using HARNESSFORGE version 1.1. These include 22 TDX Module ABI functions (6 guest-side (TDG) and 16 host-side (TDH) interface functions), listed in Table 2, resulting in 418 verification tasks.

Table 2: Results for each interface function under different havocking strategies

	Interface function	#tasks	Mem. havoc.	Obj. havoc.	Tool-spec. havoc.
Guest-side TDG	MR.REPORT	10	0	0	5
	SERVTD.WR	10	0	0	7
	SYS.RD	10	0	0	8
	VM.WR	17	0	0	14
	VP.ENTER	18	0	0	12
	VP.VMCALL	10	0	0	8
Host-side TDH	EXPORT.RESTORE	10	3	0	7
	IMPORT.ABORT	9	2	0	7
	MNG.ADDCX	33	5	2	11
	MNG.CREATE	18	3	0	17
	MNG.INIT	33	5	0	11
	MNG.KEY.CONFIG	18	4	0	16
	MNG.KEY.FREEID	17	2	0	16
	MNG.VPFLUSHDONE	21	4	0	19
	MR.FINALIZE	29	3	0	7
	PHMEM.PAGE.RECLAIM	25	0	0	5
	SYS.CONFIG	29	0	0	8
	SYS.INIT	17	0	0	13
	SYS.KEY.CONFIG	13	10	0	13
	SYS.SHUTDOWN	17	0	0	13
	SYS.UPDATE	25	0	0	19
	VP.ENTER	29	0	0	9
Overall		418	41	2	245

Execution Environment. All experiments were conducted on machines running Ubuntu 24.04 (64 bit) with Linux kernel version 6.8.0. Each machine was equipped with an Intel Xeon E3-1230 v5 CPU (3.4 GHz, 8 logical cores) and 33 GB of RAM. Each verification task was allocated 2 CPU cores, a CPU time limit of 60 min, and a memory limit of 30 GB.

5.2 Verification Results

Table 2 shows the numbers of verification tasks derived from each of the selected TDX Module interface functions and the numbers of tasks correctly solved by any of the tools, under different havocking strategies. Overall, tool-specific havocking outperformed generic byte-wise memory havocking and field-wise object havocking, making about 59 % of the generated tasks correctly solved. We further conducted several targeted analyses to understand the effects of the harness-engineering decisions on verification performance.

Havocking Strategies. Tool-specific havocking not only improved effectiveness but also the resource efficiency in terms of CPU time and memory usage. One key reason for the inefficiency of generic havocking is its reliance on potentially large loops. In contrast, tool-specific primitives allow verifiers to havoc the entire object internally in their memory model, avoiding the need to explicitly encode such loops in the tasks. Our results highlight the importance of supporting native havocking primitives in both verification-task encoding and verifier development. Notably, this insight has motivated the adoption of a standardized mechanism for symbolically havocking non-primitive data types in SV-COMP 2026.

State-Variable Constraints. About 30 % of the cover points at function entry (cf. [Sect. 4.3](#)) were unreachable by any tools. All these tasks involve preconditions that constrain the elements in the physical-address metadata table. Currently, such constraints are implemented by iterating through each element in a loop, a method that tends to be ineffective for the evaluated tools. While these element-wise constraints can be more naturally and efficiently expressed using universal quantifiers, such constructs are not commonly supported by fully automatic verification tools.

5.3 Limited Tool Performance

Despite adapting a well-established industry verification workflow (cf. [Sect. 3](#)) and exploring key harness-engineering decisions to improve performance, the overall verification success rate remains limited. In our case study, while KLEE and CBMC delivered most of the successful results (237 and 117, respectively), CPAchecker, UAutomizer, and Mopsa were unable to solve any verification tasks. Many of the failed analyses stemmed from frontend parsing (identified tool issues summarized in [Appendix B \[11\]](#)). For example, UAutomizer failed on every task due to its incomplete support for nested, anonymous `union` types, a prevalent pattern in firmware for type-punning and memory reinterpretation (more details in [Sect. 6](#)). These frontend issues prevented the tools from starting the actual verification process, indicating that frontend robustness is a fundamental limitation in applying off-the-shelf verifiers to firmware programs, despite the sophistication of their underlying algorithms.

CPAchecker similarly failed to parse 141 tasks, primarily because of anonymous `union` members within `struct` types. For the remaining tasks, CPAchecker encountered internal errors or timeouts. Upon investigation, we found that these errors were caused by the lack of support for a C extension that allows casting a scalar type to a `union` type,⁷ leading to the failure of 60 object-havocking and 27 memory-havocking tasks.

Although Mopsa’s frontend was able to parse all verification tasks, it ran into timeouts on all task variants using object havocking. For task variants using memory havocking, Mopsa managed to finish its analysis on 76 tasks but was unable to construct concrete execution paths, and hence returned `unknown` as the results.

ESBMC encountered segmentation faults or other internal errors on 264 tasks using its own havocking primitive. Most failures occurred during the encoding of verification conditions, while a few arose at an earlier program-unrolling stage.

Driven by the limited performance of the evaluated tools, we manually investigated the verification tasks and tool outputs to identify firmware-specific code patterns that hinder effective analysis and envision potential improvements to scalable verification, as explained in [Sect. 6](#).

Comparisons to Results at SV-COMP and Test-Comp 2026. The 418 verification tasks derived from TDX Module have been integrated into the 2026 editions of SV-COMP and Test-Comp. Readers interested in the performance of a wider array of tools on these tasks are referred to the official competition

⁷ <https://gcc.gnu.org/onlinedocs/gcc/Cast-to-Union.html>

reports [10, 18]. Our evaluation differs from the competition setups in three aspects: (1) Computing resources: We provided higher computing resources (Sect. 5.1) per task than the competitions. (2) Tool versions: We used SV-COMP 2025 versions for CPACHECKER, MOPSA, UAUTOMIZER, and ESBMC; for CBMC and KLEE, which are not active competition participants, we used more recent versions and updated their wrapper scripts. (3) Havocking strategies: The tasks submitted to the competitions utilize `__VERIFIER_nondet_memory()`, the newly introduced primitive for object havocking inspired by this case study. This evaluation did not include tasks using the primitive as the assessed tools had not yet supported it.

6 Bottlenecks and Opportunities

Prior research on firmware verification has primarily focused on modeling software-hardware interactions. However, as shown in our experimental results in Sect. 5, even with available hardware abstractions, applying off-the-shelf software analyzers to firmware remains far from a straightforward, push-button process. While firmware shares certain similarities with higher-level C programs, its unique characteristics often pose significant challenges on existing tools. In this section, we uncover these distinguishing features and discuss how they hinder the effective use of off-the-shelf tools. We also explore potential directions for developing new approaches tailored to the specific demands of firmware verification.

Extensive Use of Nested Type Punning. Type punning allows the reinterpretation of a data object’s memory representation as a different type. In C programs, it is usually achieved through pointer-type castings or `union` types, which allow multiple fields of different types to occupy the same memory region. It enables programmers to write data as one type and read them later via another, and vice versa.

Firmware interacts directly with hardware, writing input arguments to registers for hardware to consume and retrieving output after the operations complete. Unlike virtualized environments, only a limited, finite number of registers are available on hardware to be shared across different operations, making type punning an essential way to efficiently interface between different operational contexts in firmware (see the Appendix C.1 [11] for an example).

We found that off-the-shelf software analyzers often do not have a robust frontend to handle deeply nested, and potentially anonymous, uses of `union`. To make it worse, pointer-type casting and `union` are often used interchangeably in firmware. This can lead to misinterpretations of data structures, causing quiet incorrect verification results or tool crashes.

Explicit and Static Memory Management. Precise control over the exact memory layout of data structures is imperative for firmware. Such control is essential to ensure that variables stored in registers and data payloads in memory can be correctly processed by hardware. This is more than consistent and correct interpretation within the program level. To achieve this goal, firmware commonly uses compiler attributes to explicitly define the memory layouts, particularly for data structures that interface directly with hardware. For example, firmware uses

the attribute `aligned(n)` to specify a minimum alignment of `n` bytes, and the attribute `__packed__` to ensure no padding is added between fields. An example is provided in [Appendix C.2 \[11\]](#).

We found that off-the-shelf software analyzers often lack support for compiler attributes related to precise memory-layout control. In particular, we observed that CBMC, CPACHECKER, and UAUTOMIZER do not correctly handle these constraints (see [Appendix B \[11\]](#) for more information). This can lead to incorrect interpretation of program behavior, particularly in cases where memory accesses rely on base addresses and offsets computed under the assumption of a correct memory layout.

On the other hand, firmware’s static memory allocation also provides an opportunity for more efficient analysis. Tools that recognize such a no-heap assumption could simplify their memory models, for example, by eliminating alias analysis for heap pointers. However, we did not yet observe tools benefiting from this characteristic in our study. Before reaching a stage where memory-model simplifications would matter, the evaluated tools encountered more fundamental obstacles, such as parsing failures on nested union types and encoding errors on complex data structures, as discussed above.

Standard Libraries and System Calls. Firmware operates below the operating system and is often subject to stricter size constraints due to limited embedded memory. Unlike higher-level C programs, it lacks access to standard libraries and system calls. As a result, rather than relying on external dependencies, firmware often reimplements some selected subset of these functionalities internally.

While firmware avoids issues related to external dependencies, its reimplementation introduces an emerging obstacle: State-of-the-art software analyzers typically develop their own formal-friendly stubs for common operations in standard libraries. However, firmware’s custom reimplementations prevent these tools from identifying such functions, limiting their ability to apply built-in optimizations. Furthermore, such reimplementations may require additional effort to mock as they often include platform-specific instructions written in inline assembly. A reimplementation of the function `memcpy` in TDX Module is shown in [Appendix C.3 \[11\]](#).

To address this emerging challenge, we advocate for a standardized annotation scheme to label the reimplementations of commonly used standard-library functions, such as `memcpy`, `memset`, `memcmp`, and string functions. Such annotations would enable verification tools to reliably recognize reimplemented functions in firmware (and higher-level C programs in general) as *semantically equivalent* to their standard-library counterparts, and thus facilitate the substitution of built-in, formal-friendly models of the tools without parsing the inline assembly.

Property-Directed Slicing. Most slicing techniques implemented in state-of-the-art tools consider a given verification task homogeneously, i.e., treating preconditions (assumptions) equally as the control flow of the target function without leveraging it for slicing. Our case study highlights the opportunity for *property-directed slicing* through a clearer structure of the proof-harness design.

Each verification task consists of five components in sequence: (1) setup, (2) preconditions, (3) target function, (4) postconditions, and (5) teardown.

While components (1) and (5) depend on the target function (3), components (2) and (4) are specific to the property of each verification task. The property-under-verification, e.g., correct error-handling under invalid input arguments, can significantly influence the program paths exercised within (3), and thus the state variables required for setup and teardown in (1) and (5), respectively.

In our case study, we found property-directed slicing especially effective for negative-space verification tasks, where precondition violations lead to early returns from the target function. For example, applying this technique to an input-validation task of a guest-side interface function `TDG.VP.VMCALL` turns the task unsolvable within an hour into one solved in seconds. This orders-of-magnitude speedup demonstrates that incorporating property semantics into program slicing—beyond the syntactic dependency analysis performed by existing tools—is a promising direction for more scalable verification.

Moreover, in our case study, we often observed that many fields in a large, complex data structure are irrelevant to the property under verification, but explicitly havocking the entire object includes the irrelevant fields in the analysis, thus increasing the verification complexity unnecessarily. Therefore, finer-grained, *field-level slicing* that initializes only the accessed fields in a large, complex data structures is another important direction for future work.

7 Conclusion

Firmware underpins system security, yet its verification remains largely manual. The absence of open-source examples and tool support has long hindered the systematic use of formal methods at this layer. In this case study, we investigated how mature techniques for automatic software verification can be transferred to firmware verification. Our methodology emphasizes a clear separation between the target function and the auxiliary proof harness that constructs the verification context, together with modular, type-based management of verification artifacts. This structure enables automated extraction of verification tasks suitable for off-the-shelf analyzers and facilitates the reuse of harness components.

Our study over 418 verification tasks from 22 interface functions reveals current limitations and future opportunities: Frontend fragility remains a bottleneck in applying software analyzers to firmware. Tool-specific havocking primitives significantly improve performance, motivating SV-COMP 2026’s adoption of a standardized havocking mechanism. We further identified the potential of property-directed slicing to achieve orders-of-magnitude speedup on negative-space properties. By releasing our verification tasks, hardware abstractions, and proof harnesses, we provide realistic firmware verification challenges that enable reproducible research and future tool development.

Data-Availability Statement. The source code of TDX Module, the developed proof harnesses, and the derived verification tasks are archived on Zenodo [12]. An artifact for reproducing the results presented in Sect. 5 is also available on Zenodo [13]. In addition, a supplementary webpage is hosted at <https://www.sosy-lab.org/research/tdx-module-firmware-verification>.

Funding Statement. This project was funded by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY) and 536040111 (Bridge), and a research gift from Intel.

References

1. Intel Trust Domain Extensions. <https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html>, accessed: 2025-07-08
2. The Coq proof assistant, version 8.16.0 (2022). <https://doi.org/10.5281/zenodo.7313584>
3. Aktas, E., Cohen, C., Eads, J., Forshaw, J., Wilhelm, F.: Intel Trust Domain Extensions (TDX) security review. Tech. rep., Google Project Zero (2023). https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf
4. Alglave, J., Donaldson, A.F., Kröning, D., Tautschnig, M.: Making software verification tools really work. In: Proc. ATVA. pp. 28–42. LNCS 6996, Springer (2011). https://doi.org/10.1007/978-3-642-24372-1_3
5. Ba, J., Böhme, M., Mirzamomen, Z., Roychoudhury, A.: Stateful greybox fuzzing. In: Proc. USENIX Security. pp. 3255–3272. USENIX Association (2022). <https://www.usenix.org/conference/usenixsecurity22/presentation/ba>
6. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPACHECKER 2.3 with strategy selection (competition contribution). In: Proc. TACAS (3). pp. 359–364. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
7. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM **54**(7), 68–76 (2011). <https://doi.org/10.1145/1965724.1965743>
8. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: Proc. POPL. pp. 1–3. ACM (2002). <https://doi.org/10.1145/503272.503274>
9. Beyer, D.: Advances in automatic software testing: Test-Comp 2025. In: Proc. FASE. pp. 257–274. LNCS 15693, Springer (2025). https://doi.org/10.1007/978-3-031-90900-9_13
10. Beyer, D.: Evaluating tools for automatic software testing (Report on Test-Comp 2026). In: Proc. FASE. pp. 449–468. LNCS 16504, Springer (2026). https://doi.org/10.1007/978-3-032-22774-4_23
11. Beyer, D., Chien, P.C., Huang, B.Y., Lee, N.Z., Lemberger, T.: A case study in firmware verification: Applying formal methods to Intel® TDX Module (Appendix) (2026). <https://www.sosy-lab.org/research/tdx-module-firmware-verification/tacas26-appendix.pdf>
12. Beyer, D., Chien, P.C., Huang, B., Lee, N.Z., Lemberger, T.: The Intel TDX Module benchmark set. Zenodo (2025). <https://doi.org/10.5281/zenodo.16547223>
13. Beyer, D., Chien, P.C., Huang, B., Lee, N.Z., Lemberger, T.: Reproduction package for TACAS2026 article ‘A case study in firmware verification: Applying formal methods to Intel TDX Module’. Zenodo (2026). <https://doi.org/10.5281/zenodo.18371342>
14. Beyer, D., Grunske, L., Kettl, M., Rosenfeld, M.L., Raselimo, M.: P3: A dataset of partial program patches. In: Proc. MSR. pp. 123–127. ACM (2024). <https://doi.org/10.1145/3643991.3644889>
15. Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16

16. Beyer, D., Petrenko, A.K.: Linux driver verification. In: Proc. ISoLA. pp. 1–6. LNCS 7610, Springer (2012). https://doi.org/10.1007/978-3-642-34032-1_1
17. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS (3). pp. 151–186. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_9
18. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (Report on SV-COMP 2026). In: Proc. TACAS (2). pp. 461–502. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_23
19. Beyer, D., Strejček, J.: SV-Benchmarks: Benchmark set for software verification (SV-COMP 2025). Zenodo (2025). <https://doi.org/10.5281/zenodo.15012096>
20. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008). <https://dl.acm.org/doi/10.5555/1855741.1855756>
21. Cadar, C., Nowack, M.: KLEE symbolic execution engine in 2019 (competition contribution). Int. J. Softw. Tools Technol. Transf. **23**(6), 867 – 870 (December 2021). <https://doi.org/10.1007/s10009-020-00570-3>
22. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O’Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
23. Chong, N., Cook, B., Eidelman, J., Kallas, K., Khazem, K., Monteiro, F.R., Schwartz-Narbonne, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Code-level model checking in the software development workflow at Amazon Web Services. Softw. Pract. Exp. **51**(4), 772–797 (2021). <https://doi.org/10.1002/spe.2949>
24. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). <https://doi.org/10.1145/876638.876643>
25. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
26. Cook, B., Khazem, K., Kröning, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Model checking boot code from AWS data centers. Formal Methods Syst. Des. **57**(1), 34–52 (2021). <https://doi.org/10.1007/S10703-020-00344-2>
27. Coquand, T., Huet, G.P.: The calculus of constructions. Inf. Comput. **76**(2/3), 95–120 (1988). [https://doi.org/10.1016/0890-5401\(88\)90005-3](https://doi.org/10.1016/0890-5401(88)90005-3)
28. Costan, V., Devadas, S.: Intel SGX explained. IACR Cryptol. ePrint Arch. pp. 86:1–86:118 (2016). <http://eprint.iacr.org/2016/086>
29. Fink, X., Berger, P., Katoen, J.P.: Configurable benchmarks for C model checkers. In: Proc. NFM. pp. 338–354. LNCS 13260, Springer (2022). https://doi.org/10.1007/978-3-031-06773-0_18
30. Fox, A.C.J., Stockwell, G., Xiong, S., Becker, H., Mulligan, D.P., Petri, G., Chong, N.: A verification methodology for the Arm® Confidential Computing Architecture: From a secure specification to safe implementations. Proc. ACM Program. Lang. **7**(OOPSLA1), 376–405 (2023). <https://doi.org/10.1145/3586040>
31. Fraer, R., Keren, D., Khasidashvili, Z., Novakovsky, A., Puder, A., Singerman, E., Talmor, E., Vardi, M.Y., Yang, J.: From visual to logical formalisms for SoC validation. In: Proc. MEMOCODE. pp. 165–174. IEEE (2014). <https://doi.org/10.1109/MEMCOD.2014.6961855>
32. Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: An industrial-strength C model checker. In: Proc. ASE. pp. 888–891. ACM (2018). <https://doi.org/10.1145/3238147.3240481>

33. Geppert, T., Deml, S., Sturzenegger, D., Ebert, N.: Trusted execution environments: Applications and organizational challenges. *Frontiers Comput. Sci.* **4**, 930741:1–930741:6 (2022). <https://doi.org/10.3389/FCOMP.2022.930741>
34. Goel, A., Krstic, S., Leslie, R., Tuttle, M.R.: SMT-based system verification with DVF. In: *Proc. SMT. EPIc Series in Computing*, vol. 20, pp. 32–43. EasyChair (2012). <https://doi.org/10.29007/59RN>
35. Große, D., Kühne, U., Drechsler, R.: HW/SW co-verification of embedded systems using bounded model checking. In: *Proc. GLSVLSI*. pp. 43–48. ACM (2006). <https://doi.org/10.1145/1127908.1127920>
36. Guo, Y., Wang, Z., Zhong, B., Zeng, Q.: Formal modeling and security analysis for intra-level privilege separation. In: *Proc. ACSAC*. pp. 88–101. ACM (2022). <https://doi.org/10.1145/3564625.3567984>
37. Hania, B., Villard, M., Netzer, Y., Kodalapura, N., Nguyen, T.: Technical report of joint security review by Microsoft and Intel on Intel TDX1.5. Tech. rep., Intel Corporation and Microsoft Corporation (August 2024). https://www.intel.com/content/dam/www/public/us/en/security-advisory/documents/intel_tdx_joint_security_review_with_microsoft.pdf
38. Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüssele, F., Podelski, A.: ULTIMATE AUTOMIZER and the abstraction of bitwise operations (competition contribution). In: *Proc. TACAS* (3). pp. 418–423. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_31
39. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: *Proc. CAV*. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
40. Herdt, V., Große, D., Drechsler, R.: Enhanced Virtual Prototyping: Featuring RISC-V Case Studies. Springer (2021). <https://doi.org/10.1007/978-3-030-54828-5>
41. Horn, A., Tautschnig, M., Val, C., Liang, L., Melham, T., Grundy, J., Kroening, D.: Formal co-validation of low-level hardware/software interfaces. In: *Proc. FMCAD*. pp. 121–128. IEEE (2013). <https://doi.org/10.1109/FMCAD.2013.6679400>
42. Huang, B.Y., Ray, S., Gupta, A., Fung, J.M., Malik, S.: Formal security verification of concurrent firmware in SoCs using instruction-level abstraction for hardware. In: *Proc. DAC*. pp. 91:1–91:6. ACM (2018). <https://doi.org/10.1145/3195970.3196055>
43. Huang, B.Y., Zhang, H., Subramanyan, P., Vizel, Y., Gupta, A., Malik, S.: Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SoC) verification. *ACM Trans. Des. Autom. Electron. Syst.* **24**(1), 10:1–10:24 (2018). <https://doi.org/10.1145/3282444>
44. ISO/IEC JTC 1/SC 22: ISO/IEC 9899-2018: Information technology — Programming Languages — C. International Organization for Standardization (2018), <https://www.iso.org/standard/74528.html>
45. Jauernig, P., Sadeghi, A., Stapf, E.: Trusted execution environments: Properties, applications, and challenges. *IEEE Secur. Priv.* **18**(2), 56–60 (2020). <https://doi.org/10.1109/MSEC.2019.2947124>
46. Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: *Proc. VSTTE*. pp. 1–18. LNCS 12031, Springer (2019). https://doi.org/10.1007/978-3-030-41600-3_1
47. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: *Proc. Ershov Memorial Conference*. pp. 165–176. LNCS 5947, Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
48. Krstić, S., Yang, J., Palmer, D.W., Osborne, R.B., Talmor, E.: Security of SoC firmware load protocols. In: *Proc. HOST*. pp. 70–75. IEEE (2014). <https://doi.org/10.1109/HST.2014.6855571>

49. Kröning, D., Tautschnig, M.: CBMC: C bounded model checker (competition contribution). In: Proc. TACAS. pp. 389–391. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26
50. Lal, A., Qadeer, S.: Powering the static driver verifier using Corral. In: Proc. FSE. pp. 202–212. ACM (2014). <https://doi.org/10.1145/2635868.2635894>
51. Lee, H., Kim, S., Cha, S.K.: Fuzzle: Making a puzzle for fuzzers. In: Proc. ASE. ACM (2022). <https://doi.org/10.1145/3551349.3556908>
52. Leslie-Hurd, R., Caspi, D., Fernandez, M.: Verifying linearizability of Intel® software guard extensions. In: Proc. CAV. pp. 144–160. LNCS 9207, Springer (2015). https://doi.org/10.1007/978-3-319-21668-3_9
53. Li, X., Li, X., Dall, C., Gu, R., Nieh, J., Sait, Y., Stockwell, G.: Design and verification of the Arm Confidential Compute Architecture. In: Proc. OSDI. pp. 465–484. USENIX Association (2022). <https://www.usenix.org/conference/osdi22/presentation/li>
54. Ma, R., Wang, D., Hu, C., Ji, W., Xue, J.: Test data generation for stateful network protocol fuzzing using a rule-based state machine. *Tsinghua Science and Technology* **21**(3), 352–360 (2016). <https://doi.org/10.1109/TST.2016.7488746>
55. Ma, Y., Zhang, Q., Zhao, S., Wang, G., Li, X., Shi, Z.: Formal verification of memory isolation for the TrustZone-based TEE. In: Proc. APSEC. pp. 149–158. IEEE (2020). <https://doi.org/10.1109/APSEC51365.2020.00023>
56. Malik, S., Subramanyan, P.: Invited - Specification and modeling for systems-on-chip security verification. In: Proc. DAC. pp. 66:1–66:6. ACM (2016). <https://doi.org/10.1145/2897937.2911991>
57. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. In: Proc. HASP. ACM (2013). <https://doi.org/10.1145/2487726.2488368>
58. Miao, X., Chang, R., Zhao, J., Zhao, Y., Cao, S., Wei, T., Jiang, L., Ren, K.: CVTEE: A compatible verified TEE architecture with enhanced security. *IEEE Transactions on Dependable and Secure Computing* **20**(1), 377–391 (2023). <https://doi.org/10.1109/TDSC.2021.3133576>
59. Moloney, C., Dyer, R., Sherman, E.: Demonstrating ARG-V’s generation of realistic Java benchmarks for SV-COMP. In: Proc. TACAS (2). pp. 277–285. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_14
60. Monat, R., Ouadjaout, A., Miné, A.: MOPSA-C with trace partitioning and autosuggestions (competition contribution). In: Proc. TACAS (3). pp. 229–235. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_17
61. Mukherjee, R., Purandare, M., Polig, R., Kroening, D.: Formal techniques for effective co-verification of hardware/software co-designs. In: Proc. DAC. pp. 35:1–35:6. ACM (2017). <https://doi.org/10.1145/3061639.3062253>
62. Mulligan, D.P., Petri, G., Spinale, N., Stockwell, G., Vincent, H.J.M.: Confidential computing - A brave new world. In: Proc. SEED. pp. 132–138. IEEE (2021). <https://doi.org/10.1109/SEED51797.2021.00025>
63. Muñoz, A., Rios, R., Román, R., López, J.: A survey on the (in)security of trusted execution environments. *Comput. Secur.* **129**, 103180:1–103180:26 (2023). <https://doi.org/10.1016/J.COSE.2023.103180>
64. Nguyen, M.D., Wedler, M., Stoffel, D., Kunz, W.: Formal hardware/software co-verification by interval property checking with abstraction. In: Proc. DAC. pp. 510–515. ACM (2011). <https://doi.org/10.1145/2024724.2024843>
65. Novikov, E., Zakharov, I.S.: Towards automated static verification of GNU C programs. In: Proc. PSI. pp. 402–416. LNCS 10742, Springer (2017). https://doi.org/10.1007/978-3-319-74313-4_30

66. Ray, S., Ghosh, N., Masti, R.J., Kanuparthi, A., Fung, J.M.: Formal verification of security critical hardware-firmware interactions in commercial SoCs. In: Proc. DAC. pp. 43:1–43:4. ACM (2019). <https://doi.org/10.1145/3316781.3323478>
67. Sabt, M., Achemlal, M., Bouabdallah, A.: Trusted execution environment: What it is, and what it is not. In: Proc. TrustCom. pp. 57–64. IEEE (2015). <https://doi.org/10.1109/TRUSTCOM.2015.357>
68. Sardar, M.U., Fetzer, C.: Confidential computing and related technologies: A critical review. *Cybersecur.* **6**(1), 10:1–10:7 (2023). <https://doi.org/10.1186/S42400-023-00144-1>
69. Sardar, M.U., Faqeh, R., Fetzer, C.: Formal foundations for Intel SGX data center attestation primitives. In: Proc. ICFEM. pp. 268–283. Springer (2020). https://doi.org/10.1007/978-3-030-63406-3_16
70. Sardar, M.U., Fossati, T., Frost, S., Xiong, S.: Formal specification and verification of architecturally-defined attestation mechanisms in Arm CCA and Intel TDX. *IEEE Access* **12**, 361–381 (2023). <https://doi.org/10.1109/ACCESS.2023.3346501>
71. Sardar, M.U., Musaev, S., Fetzer, C.: Demystifying attestation in Intel Trust Domain Extensions via formal verification. *IEEE Access* **9**, 83067–83079 (2021). <https://doi.org/10.1109/ACCESS.2021.3087421>
72. Sardar, M.U., Quoc, D.L., Fetzer, C.: Towards formalization of enhanced privacy ID (EPID)-based remote attestation in Intel SGX. In: Proc. DSD. pp. 604–607. IEEE (2020). <https://doi.org/10.1109/DSD51259.2020.00099>
73. Schmidt, B., Villarraga, C., Bormann, J., Stoffel, D., Wedler, M., Kunz, W.: A computational model for SAT-based verification of hardware-dependent low-level embedded system software. In: Proc. ASP-DAC. pp. 711–716. IEEE (2013). <https://doi.org/10.1109/ASPDAC.2013.6509684>
74. Steffen, B., Isberner, M., Naujokat, S., Margaria, T., Geske, M.: Property-driven benchmark generation: Synthesizing programs of realistic structure. *Int. J. Softw. Tools Technol. Transfer* **16**(5), 465–479 (2014)
75. Subramanyan, P., Sinha, R., Lebedev, I., Devadas, S., Seshia, S.A.: A formal foundation for secure remote execution of enclaves. In: Proc. CCS. pp. 2435–2450. ACM (2017). <https://doi.org/10.1145/3133956.3134098>
76. Westhofen, L., Berger, P., Katoen, J.P.: Benchmarking software model checkers on automotive code. In: Proc. NFM. pp. 133–150. LNCS 12229, Springer (2020). https://doi.org/10.1007/978-3-030-55754-6_8
77. Wu, P., Shen, Q., Deng, R.H., Liu, X., Zhang, Y., Wu, Z.: OblIDC: An SGX-based oblivious distributed computing framework with formal proof. In: Proc. Asia CCS. pp. 86–99. ACM (2019). <https://doi.org/10.1145/3321705.3329822>
78. Wu, T., Li, X., Manino, E., Menezes, R., Gadelha, M., Xiong, S., Tihanyi, N., Petoumenos, P., Cordeiro, L.: ESBMC v7.7: Efficient concurrent software verification with scheduling, incremental SMT and partial order reduction (competition contribution). In: Proc. TACAS (3). pp. 223–228. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_16
79. Wu, T., Xiong, S., Manino, E., Stockwell, G., Cordeiro, L.C.: Verifying components of Arm[®] Confidential Computing Architecture with ESBMC. In: Proc. SAS. pp. 451–462. LNCS 14995, Springer (2024). https://doi.org/10.1007/978-3-031-74776-2_18

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Extending FRET with SLEEC Rules: Formalization, Obligation Inference, and Monitoring*

Mahrokh Mirani , Paola Inverardi ,
Patrizio Pelliccione , Franco Raimondi ,
and Nicolas Troquard

Gran Sasso Science Institute, L'Aquila, Italy
{mahrokh.mirani, paola.inverardi,
patrizio.pelliccione, franco.raimondi,
nicolas.troquard}@gssi.it

Abstract. The increasing adoption of AI and autonomous systems has raised concerns about their potential negative impacts. To ensure that these systems align with human values, methodologies have been proposed to elicit SLEEC (Social, Legal, Ethical, Empathetic, and Cultural) requirements. In this paper, we present an extension of FRET, a widely used and industry-accepted tool developed by NASA, with the ability to manage SLEEC rules as formal requirements. In addition to implementing the ability to manage the lifecycle of SLEEC rules in FRET, we also provide a mechanism to generate an inference engine for SLEEC obligations and a set of run-time monitors based on Datalog. We validate our proposal on 9 use cases from the literature that include 227 SLEEC rules, demonstrating its ability to manage them. We also assess the performance and scalability of both the obligation inference engine and the monitors, demonstrating the ability of Datalog to scale to large scenarios.

Keywords: Social, Legal, Ethical, Empathetic, and Cultural rules (SLEEC) · Formal Requirement Elicitation Tool (FRET) · Normative Requirements · Requirement Management.

1 Introduction

The adoption of autonomous systems and AI is increasingly posing risks to humans, society, and the environment [9]. This is testified by the flourishing of guidelines and recommendations [34, 35, 24], laws and regulations [22], as well as standards (more than 25 new standards in the last years plus extensions of existing ones) that try to regulate and alleviate the lack of good practices for the governance, management, and quality of AI systems [25].

As a consequence, ethical considerations in autonomous systems have gained significant attention in recent years [29, 19]. *Machine ethics* [17], which is a combination of computational logic and moral philosophy [30, 5] is one of the

* Corresponding author: Mahrokh Mirani

outcomes of the research community. Other approaches try to integrate ethical values during design and development [4, 28]. Other researchers tried to elicit and model the ethical values of individual users through surveys [3, 2]. Indeed, it is recognized that ethical values also have a subjective dimension caused by context, culture, and individual perspectives [18, 6, 4, 7].

Recently, [32] proposed a method to translate norms and normative principles, which are, typically, of a social, legal, ethical, empathetic, or cultural ('SLEEC') nature, into explicitly formulated practical rules. SLEEC rules were introduced to serve as an instrument to ensure that autonomous agents follow the Asilomar AI Principles, and in particular that "highly autonomous AI systems should be designed so that their goals and behaviours can be assured to align with human values through their operation" [15]. More details about AI value alignment can be found in [26]. Following [32], several contributions addressed the refinement of high-level principles into lower-level operational rules, and the resolution of conflicts and other properties such as redundancy between normative principles [11, 13, 33, 38, 31]. However, these approaches work at design time, with the aim of solving potential normative conflicts and enable the design of agents able to implement decisions involving normative principles. In realistic scenarios, there is also the need for support in reasoning about and verifying SLEEC aspects encountered at runtime in the field.

To better explain our contributions, we introduce a running example in which a robot must greet people at an assistance desk.¹ A SLEEC rule in this scenario can be defined to respect human dignity. This rule will be defined as follows: If the robot sees a person (A_0), THEN the robot should say "Hi" (B_0), UNLESS the person is deaf (A_1), IN WHICH CASE the robot should wave at the person (B_1), UNLESS the person is blind (A_2), IN WHICH CASE the robot should tap the shoulder of the person (B_2). This rule is designed consistently with other rules, and there are no conflicts or redundancies. However, already in this simple example, it becomes clear that there can be various alternatives, and at runtime, only one of them will be activated. So, ethical problems, impacting the human dignity of an individual in this case, may show up at runtime when the robot will meet specific persons and may not greet them properly. The translation of this requirement into an actual implementation requires the ability of reasoning about obligations at runtime, possibly in a very large system.

Our aims are (1) to provide a FRET extension to solve the issue of managing SLEEC requirements and (2) to generate obligation inference engines with associated monitors that can scale to large instances. Given the existing research background, in our work we assume that SLEEC rules have no conflicts, inconsistencies, and are realizable. Specific details about these checks can be found in several other works, such as [38, 31].

FRET [20] is a widely-used and industry-accepted tool developed by NASA, often used to manage the lifecycle of requirements, typically with the aim of certi-

¹ The example is intentionally simple and stereotypical to help with intuition; we refer the reader to Section 5 for more realistic use cases and to Section 3 for its formalization.

flying software for safety-critical missions: our extension enables the management of SLEEC rules and can thus be employed, for example, to demonstrate compliance with external regulations, such as the EU AI act or other regulations [22, 35, 24]. Then, we automatically synthesize an obligation inference engine and runtime monitors that can be used to check whether the behavior of the system is compliant with the specified SLEEC rules. We validate our FRET extension on 9 use cases from the literature that include 227 SLEEC rules, demonstrating its ability to manage them and assessing the performance and scalability of both the obligation inference engine and the monitors.

In summary, the main contributions of the paper are as follows.

- An extension of FRET with the ability to manage the lifecycle of SLEEC rules as formal requirements.
- The automatic synthesis of obligation inference engines for SLEEC rules.
- The automatic synthesis of run-time monitors.
- Proofs of correctness and complexity.
- An evaluation of performance and scalability that demonstrates how the Datalog-base approach can scale to large sets of rules.

The paper is organized as follows: Section 2 provides background information on SLEEC rules and other techniques that are employed in the paper. Section 3 explains the syntax and semantics of our FRET extension and Section 4 describes the generation of the obligation inference engine and of runtime monitors, providing correctness and complexity results. Section 5 describes the validation of our FRET extension. Section 6 discusses the related works; the paper concludes with final remarks and future research directions in Section 7.

2 Background

This section presents background material to situate our work. We review prior efforts on SLEEC rule formalization and two existing interpretations of these rules. We also introduce FRET and Datalog as the technical foundations for our approach.

2.1 SLEEC Rules

Townsend et al. [32] present a methodology for domain experts, ethicists, lawyers, etc., to elicit normative requirements. The resulting formulation consists of SLEEC rules. A SLEEC rule σ is elicited in natural language, but follows a predictable pattern:

IF A_0 THEN B_0 ,
 UNLESS A_1 IN WHICH CASE B_1 ,
 UNLESS A_2 IN WHICH CASE B_2 ,
 ...
 UNLESS A_n IN WHICH CASE B_n .

SLEEC rules are not intended to serve as a formal language, but are rather natural language expressions derived from domain expertise—by experts with no logical training—through a principled elicitation methodology. As such, some ambiguity may persist, which can only be resolved through mutual agreement among stakeholders on the precise semantics. It is therefore crucial to acknowledge this inherent ambiguity and ensure that tools are capable of handling multiple interpretations.

There exist two different interpretations of SLEEC rules in the literature. One interpretation comes from [33], and we will call it the *L'Aquila semantics*, or *AQ-semantics*. The other interpretation comes from [11, 38], and we will call it the *York-Toronto semantics*, or *YT-semantics*. The AQ- and YT-semantics differ in the interpretation of the associativity of the words and groups of words.

In the AQ-semantics, the previous SLEEC rule σ is read as IF A_0 THEN (B_0 , UNLESS A_1 IN WHICH CASE (B_1 , UNLESS A_2 IN WHICH CASE (B_2 , ... UNLESS A_n IN WHICH CASE (B_n) ...))). It can be captured in classical logic by the following formula:

$$\phi_{AQ}(\sigma) = \left[\bigwedge_{0 \leq i \leq n-1} \left(\left(\bigwedge_{0 \leq j \leq i} (A_j) \wedge \neg A_{i+1} \right) \rightarrow B_i \right) \right] \wedge \left[\left(\bigwedge_{0 \leq j \leq n} (A_j) \right) \rightarrow B_n \right] \quad (1)$$

In the YT-semantics, the same SLEEC rule σ is read as IF A_0 THEN (... ((B_0 , UNLESS A_1 IN WHICH CASE B_1), UNLESS A_2 IN WHICH CASE B_2), ... UNLESS A_n IN WHICH CASE B_n). It can be captured in classical logic by the following formula:

$$\phi_{YT}(\sigma) = \left[\bigwedge_{0 \leq i \leq n} \left(\left(A_0 \wedge A_i \wedge \bigwedge_{i < j \leq n} (\neg A_j) \right) \rightarrow B_i \right) \right] \quad (2)$$

These two translations into logic formulas are simple consequences of respecting the parentheses and translating the clauses ‘ A UNLESS B IN WHICH CASE C ’ as $(\neg B \rightarrow A) \wedge (B \rightarrow C)$ justified in [33]. (‘IF A THEN B ’ is simply translated as $A \rightarrow B$.) Moreover, $\phi_{YT}(\sigma)$ can also be equivalently obtained by applying the semantics proposed by [11]. To illustrate the difference in semantics, we turn back to our example. Assuming that the person is blind but not deaf (i.e., A_0, A_2 are true, but A_1 is false), in the AQ-semantics the robot would say “Hi”, and in the YT-semantics the robot would tap the shoulder.

The two semantics are equivalent in terms of expressivity and computational complexity. Nevertheless, their distinct existence is undeniable, and we therefore chose to keep both in the FRET extension as each may be more intuitive to different users. The key consideration is that the selected semantics be agreed upon within the team to prevent potential misunderstandings. In this regard, using a

tool like FRET that can keep track of updates and refinements of requirements can be highly beneficial.

2.2 Formal Requirement Elicitation Tool

Formal Requirements Elicitation Tool (FRET) is a requirements engineering tool [16] developed by the NASA Ames Research Center. It serves to specify requirements of complex safety-critical systems. A DSL is used as an intermediate language between stakeholders and ‘knowledge engineers’. It is a controlled natural language called FRETish. FRET allows for the automated conversion of FRETish requirements into temporal logic formulas. FRET is also designed with a user-friendly interface that helps in eliciting, evolving, and maintaining the requirements. Therefore, it is used in industry for managing the requirements during the software life cycle.

Adapting FRET to handle normative requirements enables automatic translation of them into logic programs. This leads to correct-by-construction obligation inference engines and monitors for normative decision-making in critical systems.

2.3 Datalog

In this work, we employ Datalog [1, 10], and in particular the Datalog solver Soufflé [27], to reason about SLEEC rules. Datalog is a declarative programming language, a decidable fragment of Prolog that is guaranteed to terminate. As detailed below, in our work we employ an even more limited fragment of Datalog without recursion and without complex operators. We refer to [27] for additional details.

3 Syntax and Semantics

In this section, we explain the syntax and semantics of SLEEC-SLANG (SLEEC Specification Language), a language we defined to encode SLEEC rules in FRET. We also compare this language to the existing literature and explain how requirements in other formalisms can be translated into SLEEC-SLANG.

3.1 Syntax of SLEEC-SLANG

In this section, we present the syntax of SLEEC DSL [38], and we map it to the syntax of SLEEC-SLANG. An existing tool for specification, validation and verification of SLEEC requirements is SLEECVAL [38]. This tool uses SLEEC DSL, a language specifically developed for formalisation of SLEEC requirements. The authors of SLEECVAL provide the syntax of a domain-specific language for specifying SLEEC rules and define semantics in tock-CSP process algebra. Table 1 presents an expanded fragment of the syntax of this language. We made datatypes in the rules explicit to be able to capture them in SLEEC-SLANG.

```

rule ::= ruleID 'when' trigger 'then' response
trigger ::= eventID | eventID 'and' trigBoolExpr
trigBoolExpr ::= mBoolExpr | 'not' trigBoolExpr | (trigBoolExpr) | trigBoolExpr boolOp trigBoolExpr
mBoolExpr ::= boolMeasureExpr | numMeasureExpr
boolMeasureExpr ::= '{' boolMeasureID '}'
numMeasureExpr ::= '{' numMeasureID '}' relOp value
response ::= constraint defeaters?
constraint ::= eventID
defeaters ::= defeater | defeater defeaters
defeater ::= 'unless' mBoolExpr ('then' response)?
relOp ::= '<' | '>' | '<>' | '<=' | '>=' | '='
boolOp ::= 'and' | 'or'

```

Table 1: Part of the syntax of SLEEC DSL with explicit types.

```

Specification ::= ('SLEEC')? 'IF' trigger 'THEN' action ('UNLESS' defeater 'IWC' obligation)*
trigger ::= cond_expression
defeater ::= cond_expression
action ::= obl_expression
obligation ::= obl_expression
cond_expression ::= '! cond_expression | cond_expression '&' cond_expression | cond_expression '|'
cond_expression | '(' cond_expression ')' | predicate
obl_expression ::= obl_expression '&' obl_expression | '(' obl_expression ')' | predicate
predicate ::= atomic_predicate | 'true' | comparison_predicate
atomic_predicate ::= predicate_name '(' ( (var_name(':' 'number')? ) (',' var_name(':' 'number')?
)*)? ')'
predicate_name ::= ID
comparison_predicate ::= var_name relational_op var_name
var_name ::= ID
relational_op ::= '<' | '<=' | '>' | '>=' | '=' | '!'
ID ::= ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*

```

Table 2: Syntax of SLEEC-SLANG.

The syntax of SLEEC-SLANG is presented in Table 2. A rule can start with the keyword “SLEEC” but it is not mandatory. Then, an “IF” is followed by a predicate expression and the “THEN” keyword. This is the trigger for the rule. The obligation corresponding to this trigger is specified after the “THEN” keyword. This default rule can then be followed by arbitrarily many (unbounded but finite) defeaters, specifying exceptions and their consequences. These defeaters are specified using the keywords “UNLESS” and “IWC” (the latter stands for “in which case”). All conditions and obligations are expressions of predicates with zero or more arguments. However, it is not allowed to have a disjunction in an obligation. It is also assumed that the set of predicates for triggers and defeaters is disjoint from the set of predicates for obligations. It is possible to transform a requirement from SLEEC DSL into SLEEC-SLANG by substituting certain keywords and applying syntactic modifications so that SLEEC DSL requirements can be directly encoded in SLEEC-SLANG, as demonstrated in Section 5. This translation is explained in more detail in Appendix A.

Proposition 1. *A SLEEC DSL requirement r of size n where n is the number of symbols in r , is translated into a SLEEC-SLANG requirement in time $O(n)$. The size of the resulting requirement is $O(n)$.*

Proof. The translation iterates over syntactic elements of SLEEC DSL in a forward pass, without recursion. All the loops are local to events or defeaters and the size of the generated strings is linearly bounded.

3.2 Semantics

The semantics of a SLEEC-SLANG rule σ inherits the meaning of the formula resulting from its translation. As explained in Section 2, each of the semantics can be chosen depending on which translation is selected. These two semantics coincide when there are zero or one defeater. They cease to agree in general in cases where there are multiple defeaters. In this section, we formally define the central notions involved in the correctness of the approach. We also illustrate the two semantics through the Example 1.

Definition 1 (Observation set). *Let Γ be a set of SLEEC-SLANG rules, let P be the set of predicates in the `cond` expressions of the rules in Γ , let F be a set of facts (instantiated input predicates), and let C be the set of constants occurring in F . We note $\text{ground}(\Gamma, F)$ the set of ground atoms obtained by instantiating predicates in P with constants in C . The set $\Omega = F \cup \{\neg A \mid A \in \text{ground}(\Gamma, F) \setminus F\}$ is the observation set resulting from Γ and F .*

Given a set of formulae Φ , the logical entailment $\Phi \models \psi$ is defined as usual: it is true when all models of Φ are also models of ψ .

Definition 2 (Entailment). *Let S be a semantics in $\{AQ, YT\}$. Let Γ be a set of SLEEC-SLANG rules. Let also Ω be an observation set, and B be an obligation. We say that B is entailed by Γ given Ω when: $\bigcup_{\sigma \in \Gamma} \phi_S(\sigma), \Omega \models B$.*

A semantics also allows one to decide when a situation violates an obligation that is entailed. To this end, for each obligation B_i we introduce a new predicate Obs_{B_i} . These predicates are Boolean and encode the fact that the action corresponding to obligation B_i was performed (for instance, in our running example, sensing that the robot said "Hi"). This set is disjoint from the other predicates. Formally:

Definition 3 (Violation). *Given a semantics S , a set of SLEEC-SLANG rules Γ , an observation set Ω and the observation Obs_B of an action corresponding to obligation B , we say that B is violated if obligation B is entailed by Γ as defined in Definition 2, but Obs_B is not entailed.*

This formalisation allows us to explore further the differences between AQ- and YT-semantics. Let us suppose that the trigger of the rule holds. Both semantics agree that the priority of defeaters introduced by ‘unless’ have increasing priority. YT-semantics enforces that the obligation corresponding to the deepest true defeater is true. A consequence of it is that if the first defeater does not hold, but the last defeater holds, then the last obligation is entailed. AQ-semantics enforces that the obligation corresponding to the last true defeaters in an initial sequence of true defeaters is true. A consequence is that if the first defeater does

not hold, then the default obligation is entailed, regardless of whether other defeaters hold. Intuitively, AQ-semantics proceeds top-down and stops at the first true defeater before a false one; YT-semantics proceeds bottom-up and stops at the first true defeater from the bottom.

Example 1. Consider the following SLEEC-SLANG requirement σ corresponding to our running example.

SLEEC IF $See(X)$ THEN $SayHi(X)$,
 UNLESS $Deaf(X)$ IWC $Wave(X)$,
 UNLESS $Blind(X)$ IWC $TapShoulder(X)$

Suppose that the trigger $See(person)$ holds. The following table indicates for both semantics which obligation is entailed depending on the truth value of $Deaf(person)$ and $Blind(person)$.

$Deaf$	$Blind$	AQ	YT
0	0	$SayHi$	$SayHi$
0	1	$SayHi$	$TapShoulder$
1	0	$Wave$	$Wave$
1	1	$TapShoulder$	$TapShoulder$

In particular, concerning the AQ-semantics, we have $\phi_{AQ}(\sigma), \{See(person), \neg Deaf(person), Blind(person)\} \models SayHi(person)$. Under the same conditions, $TapShoulder(person)$ and $Wave(person)$ are not entailed. Using the YT-semantics, $SayHi(person)$ is not entailed, and neither is $Wave(person)$. But $TapShoulder(person)$ is entailed.

From a technical viewpoint, this does not affect the complexity or the expressiveness of the requirements. In SLEEC-SLANG we support both semantics.

4 Automatic Generation of Obligation Inference Engines and Monitors

In this section, we present the problems of obligation inference and monitoring. Both are parameterised with a semantics $S \in \{AQ, YT\}$.

Definition 4 (Obligation S -Inference Problem). Let S be a semantics in $\{AQ, YT\}$. The problem of obligation S -inference is defined as follows. Given a set of SLEEC-SLANG rules Γ , an observation set $A_1(\mathbf{c}_1), \dots, A_n(\mathbf{c}_n)$, and an obligation $B(\mathbf{c}_0)$, is $B(\mathbf{c}_0)$ entailed by Γ given $A_1(\mathbf{c}_1), \dots, A_n(\mathbf{c}_n)$ when using the S -semantics?

Definition 5 (S -Monitoring Problem). Let S be a semantics in $\{AQ, YT\}$. The problem of S -monitoring is defined as follows. Given a set of SLEEC-SLANG rules Γ , an observation set $A_1(\mathbf{c}_1), \dots, A_n(\mathbf{c}_n)$, and an obligation $B(\mathbf{c}_0)$ with its corresponding observation $Obs_{B(\mathbf{c}_0)}$, is $B(\mathbf{c}_0)$ violated when using the S -semantics?

```
.decl See (X: symbol) .decl Deaf (X: symbol) .decl Blind (X: symbol) .decl
  SayHi (X: symbol) .decl Wave (X: symbol) .decl TapShoulder (X: symbol)

SayHi(X) :- (See(X)), !(Deaf(X)).
Wave(X) :- (See(X)), (Deaf(X)), !(Blind(X)).
TapShoulder(X) :- (See(X)), (Deaf(X)), (Blind(X)).

.input See, Deaf, Blind
.output SayHi, Wave, TapShoulder
```

Fig. 1: Automatically generated Datalog code for the obligation inference, AQ-semantics. The full and unaltered code is presented in Appendix B.

```
TapShoulder(X) :- (See(X)), (Blind(X)).
Wave(X) :- (See(X), !(Blind(X)), (Deaf(X)).
SayHi(X) :- (See(X)), !(Blind(X)), !(Deaf(X)).
```

Fig. 2: Automatically generated Datalog rules for the obligation inference, YT-semantics.

For each problem, we also provide a translation from SLEEC-SLANG to Datalog and demonstrate the correctness.

4.1 Synthesis of Obligation Inference Engines

To generate the Datalog code for the obligation inference, one of the logical compilations of Formula 1 and Formula 2 is used, depending on the chosen semantics. In addition to the default obligation, one Datalog rule is generated for every defeater. If all the conditions that appear in that rule are satisfied, then this obligation gets the value “true”.

To compute the conditions, observations are defined as input data to the Datalog program. This information should be provided by the system, e.g., through sensors. The computed obligations are the output data of the program. As an example, the Datalog code corresponding to the AQ-semantics for the running example is shown in Figure 1 and the equivalent YT-semantics rules are shown in Figure 2.

Proposition 2. *The Datalog obligation inference engine solves the problem of Obligation S-Inference.*

Proof. Let Γ be a set of SLEEC-SLANG rules, Ω be an observation set, and B be an obligation. We prove that Datalog outputs true for the Datalog relation B iff B is entailed by Γ given Ω . The observation set Ω is encoded directly in Datalog through input relations, with a 1-to-1 correspondence between symbols. The set of rules in Γ is translated using either the AQ-semantics (Equation 1) or the YT-Semantics (Equation 2), where the head is the corresponding Datalog obligation for obligation B and the Boolean formula is encoded verbatim simply by replacing the syntactical encoding of the operators with the Datalog equivalent symbols. An output "true" is generated exactly when the tail is satisfied. Moreover, since Ω is an observation set, it contains a complete description of the

```

SayHi(X) :- (See(X)), !(Deaf(X)).
Wave(X) :- (See(X)), (Deaf(X)), !(Blind(X)).
TapShoulder(X) :- (See(X)), (Deaf(X)), (Blind(X)).
violation_of_SayHi(X) :- (!observed_SayHi(X), SayHi(X)).
violation_of_Wave(X) :- (!observed_Wave(X), Wave(X)).
violation_of_TapShoulder(X) :- (!observed_TapShoulder(X), TapShoulder(X)).

```

Fig. 3: Automatically generated Datalog rules for the runtime monitor, AQ-semantics.

input symbols. Therefore, the entailment relation for this coincides with that of classical Boolean entailment.

4.2 Synthesis of Monitors

Intuitively, for generating monitors, we perform obligation inference first (see Definition 2). Then, we compare the result with actual observations of actions and raise an alert if these two do not match.

Formally, we introduce additional relations corresponding to the observation of actions as detailed in Definition 3. An output variable is defined for every obligation to be monitored and it is set to true if and only if the obligation has value “true”, but the observed obligation has the value “false”. The violation of obligations is then returned as an output. To illustrate this mechanism, Figure 3 report the AQ-monitor of the running example.

Proposition 3. *The Datalog monitoring engine solves the problem of S-Monitoring.*

Proof. Consider the definition of the violation of an obligation as defined in Definition 3. The Datalog rule to encode this violation simply records the difference (if any) between an observed event and the entailed obligation. Correctness follows from Proposition 2.

4.3 Complexity

We demonstrated the correctness of our approach to solve the problems of obligation inference and monitoring. We now turn our attention to the complexity of solving these problems.

First, we observe that the translations from SLEEC-SLANG into Datalog engines can be done efficiently.

Proposition 4. *A SLEEC-SLANG requirement of size n with m defeaters is translated into a Datalog obligation inference program in time $O(n(m + 1))$. The size of the resulting program is $O(n(m + 1))$. The same is true about for the Datalog obligation monitor. This is true for both AQ- and YT-semantics.*

Proof. Writing the declarations, input and output statements requires only linear time. Both semantics yield the same statements. To translate the Datalog rules, the two semantics differ. But both semantics involve writing one Datalog rule for the trigger and one rule for every defeater, that is $m + 1$ rules. Each of these rules has a size that is clearly bounded by the size of the requirement, that is n . The Datalog engine of the corresponding monitor has $m + 1$ additional rules of constant size.

We are ready to state the complexity of the problems of obligation inference and monitoring.

Proposition 5. *The problems of Obligation S -Inference and S -Monitoring can be solved in exponential time with respect to the size of the SLEEC-SLANG requirements. The complexity can be solved in polynomial time with respect to the amount of data.*

Proof. Proposition 2 and Proposition 3 state that we can reduce the problems of Obligation S -Inference and of S -Monitoring to querying the Datalog obligation inference and monitoring engines. To decide whether an obligation or violation $B(\mathbf{c})$ is entailed from a set of SLEEC-SLANG requirements and the data, it suffices to query $B(\mathbf{c})$ over the data and the corresponding Datalog engine. It is PTIME in terms of data complexity and EXPTIME in terms of expression complexity [10]. The result follows directly because of Proposition 4.

This means that normative decision making using SLEEC rules can be implemented in a way that scales with the amount of data that the system accumulates through time.

5 Implementation and Validation in FRET

In this section we describe the implementation of our proposal in FRET. The generation of the obligation inference engine and the generation of monitors is executed upon successful parsing of a SLEEC rule. This generation is performed through a tree walker over the parse tree and is contextual with the JavaScript editor.

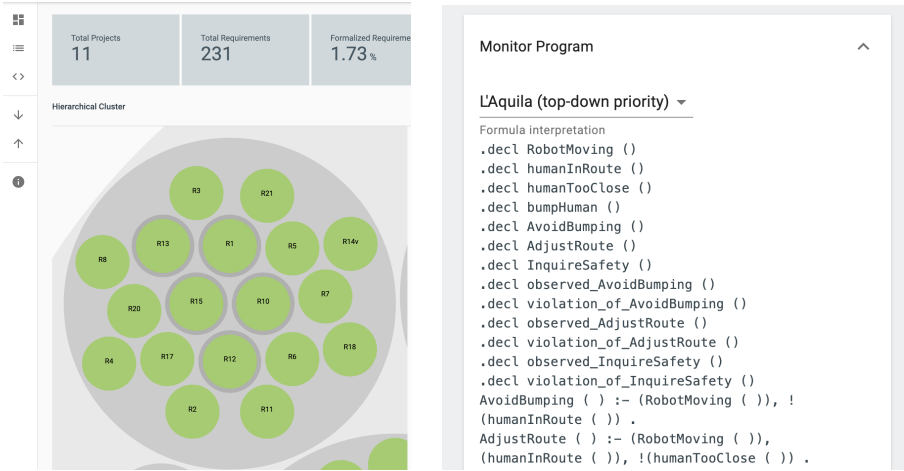
For the validation, we are interested with the tools' ability to apply to the case studies, but we also offer an evaluation of the performance of the synthesised obligation inference engines and monitors. In particular, we assess performance by generating random instances of obligation inference or monitoring tasks, and we assess scalability by measuring the time required to infer obligations or to perform monitoring over an increasing number of rules with random input.

5.1 Rules Datasets

To evaluate our FRET extension, we consider the use cases that have been formalized in [12]. This GitHub repository contains the formalization of 9 case studies: Ambient Assisted Living for Long-term Monitoring and Interaction (ALMI);

Autonomous Systems for Forest Protection (ASPEN); Autonomous Cars (Auto-Car); Body Sensor Network (BSN); Confident Safety Integration for Collaborative Robots (CSI:Cobot); Diagnostic AI System for Robot-Assisted ED Triage (DAISY); Data Processing Agreement (DPA); Assisted Care Dressing (DressAssist); Safe of Shared Control in Autonomous Driving (safeSCAD).

Each use case is formalized in SLEEC DSL and we translated it to SLEEC-SLANG as described in Appendix A. The original dataset has a total of 279 rules: we did not encode 15 rules involving timing constraints (currently not supported in our extension) and we discarded 37 rules where the appearance of the **otherwise** keyword was not matched by a **within** scope as required by SLEEC DSL, obtaining 227 rules. The rules can be imported by selecting the file `caseStudies.json` in the folder `caseStudies/FRETishSLEEC/` of our tool. Figure 4 displays the main window of the extension, and the automatically generated monitor for rule R1 of case study CSI. Notice that the semantics (AQ vs. YT) can be selected at the top of the generated Datalog code.



(a) Main FRET window for use case CSI (b) Automatically generated monitor for Rule 1 of CSI.

Fig. 4: Screenshots of FRET user interface for SLEEC-SLANG requirements.

5.2 Performance Evaluation

For the elementary assessment of the performance of the obligation inference engine and of the monitors, we have generated a random test for each semantics (AQ or YT) and for each activity (obligation inference or monitor), resulting in 908 overall tests, which are available in the archive. All tests were carried out on a Mac machine with an 8-core M1 Pro chip, 16 GB of RAM, running macOS Sequoia (15.2), and Soufflé (2.4.1) as the Datalog logic engine.

Table 3: Average, max, and min execution times and standard deviation for type obligation (obl) or monitoring (mon), for each case study.

Case Study	Type	Average (s)	Max (s)	Min (s)	std. dev.
ALMI	obl	0.018	0.046	0.015	0.0044
	mon	0.018	0.032	0.016	0.0032
ASPEN	obl	0.016	0.021	0.015	0.0011
	mon	0.017	0.023	0.016	0.0015
Autocar	obl	0.017	0.019	0.015	0.0010
	mon	0.017	0.021	0.016	0.0011
BSN	obl	0.016	0.020	0.016	0.0009
	mon	0.017	0.022	0.015	0.0013
CSI	obl	0.016	0.020	0.016	0.0009
	mon	0.017	0.020	0.016	0.0010
DAISY	obl	0.016	0.019	0.015	0.0007
	mon	0.017	0.019	0.016	0.0007
DRESSASSIST	obl	0.016	0.019	0.015	0.0010
	mon	0.017	0.021	0.016	0.0013
GDPR	obl	0.017	0.021	0.016	0.0010
	mon	0.017	0.023	0.016	0.0016
safescade	obl	0.016	0.023	0.015	0.0012
	mon	0.017	0.021	0.016	0.0009

Execution times are reported in Table 3. Overall, the average time for the computation of an obligation is 0.016 seconds, while the average time monitoring obligations is only slightly larger (0.017 seconds), which is consistent with the fact that monitors have more rules to compute. All execution times are in the range 0.015–0.046 seconds, indicating that systems using SLEEC rules can make normatively correct decisions and can be monitored efficiently at runtime.

5.3 Scalability Evaluation

To assess the scalability of Datalog, we considered the 227 rules and we built an increasingly large Datalog instance in the following way: considering the rules in alphabetical order, we measured the time required to perform obligation inference under the AQ semantics for the obligation inference task. We then added to the same datalog file the monitoring tasks. Iteratively, we added all the other rules. Finally, we repeated the process for the YT semantics, generating 908 different Datalog files of increasing complexity, using randomly generated input facts. Execution time for these files are reported in Figure 5.

Overall, execution times ranged from less than 0.02 seconds for a single problem instance to approximately 6 seconds for 908 problems. The file encoding 908 problems has a length of 12020 lines and includes approximately 3600 different predicates, 2600 of which are input predicates, and 1000 output predicates. For the largest instance, there are 8221 input facts, for a total of approximately 6 Mb.

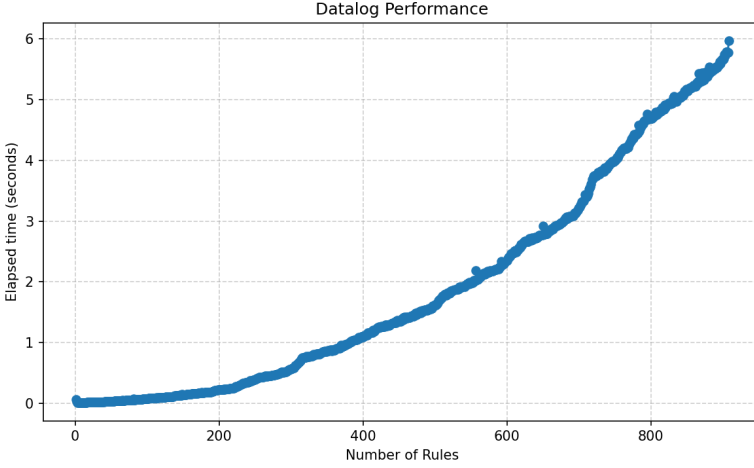


Fig. 5: Scalability results.

This scalability result demonstrates that a Datalog-based approach can operate on commodity hardware even in the presence of multiple and simultaneous rules, on a scenario that is comparable in size to real-world applications.

6 Related Work

Normative requirements are important in modern AI and software. They are critical properties that software must satisfy. As such, we must engineer them with at least the same care as we engineer requirements for the most critical systems traditionally encountered in manufacturing, healthcare, avionics, etc.

Table 4: A comparison between different SLEEC requirement tools

Tool	Conflict Checking	Redundancy Checking	Restrictiveness Checking	Insufficiency Checking	Monitor Generation	Obligation Inference
LEGOS-SLEEC [11]	✓	✓	✓	✓	–	–
AUTOCHECK [14]	✓	✓	–	–	–	–
SLEECVAL [37]	✓	✓	–	–	–	–
FRETishSLEEC	–	–	–	–	✓	✓

Several tools have already been proposed to work with SLEEC rules. Concerning the validation of a set of SLEEC rules, SLEEC-LEGOS [11, 21] is a tool to help elicit the normative non-functional requirements into SLEEC rules. In this respect, it is very similar to SLEECVAL [38]. The specification languages

are also similar, and both tools support the YT-semantics. As a distinguishing feature, SLEEC-LEGOS is based on a variant of First-Order Logic, while SLEECVAL is based on the process algebra tock-CSP. Our work is orthogonal, in that it assumes the correctness of SLEEC rules, and it generates an obligation inference engine and monitors as part of a tool to manage the lifecycle of requirements. The differences between these tools are summarized in Table 4.

The work in [31] aims to understand how normative conflicts can be mapped to choices of decision-making processes through human–autonomous agent interaction, and could potentially be related to FRET. We leave this line of research open for future work.

SLEEC DSL [37] is the language used in SLEECVAL and SLEEC-LEGOS for writing SLEEC rules. Extending the original SLEEC rules from [32], SLEEC DSL also supports timed deadlines (**within** statements), that we abstract away in the present work. In our work, we start from SLEEC rules expressed in SLEEC DSL for the original SLEEC semantics, and we leave timing aspects for future work.

In terms of logic programming, [23] proposes an encoding of SLEEC rules into Datalog, following the YT-semantics, and provides some evidence that it can serve as a solution for the computation of obligations arising from SLEEC rules. This solution is also scalable. Normative deliberation is likely to happen on-board, in devices with limited resources. However, this approach is fully manual and no proof of correctness is provided.

7 Conclusion

In this paper we introduced a FRET extension that provides an obligation inference engine and a runtime monitoring engine for SLEEC requirements. We demonstrated the correctness and complexity of our approach, and presented an extensive evaluation of performance on a state of the art normative requirement dataset. Additionally, we added the possibility of modeling SLEEC requirements directly in FRET and automatically generating Datalog code for these tasks.

Our exposition focused on handling the normative rules. But working with Datalog, applications can have readily access to powerful knowledge representation and reasoning techniques. Datalog can be used to provide a background theory about the predicates that are used in the SLEEC rules. For instance, in the running example, it could be that $See(X)$ is not primitive. It could be that it is only derived from a (non SLEEC) rule $See(X) :- SensedWith(X,Y), VisualSensor(Y)$. Datalog can also handle more involved definitions involving recursion. For instance, suppose that instead of $See(X)$, the trigger in the running example is $Reachable(X)$ meaning that the robot is able to reach the person X to greet them. Suppose also that the physical environment can be represented as a graph with relation $Edge$ that the robot can navigate. Datalog can capture $Reachable(X)$ with the base rule $Reachable(X,Y) :- Edge(X,Y)$, and inductive rule $Reachable(X,Z) :- Edge(X,Y), Reachable(Y,Z)$. Such a background knowledge can be imported in the form of W3C OWL2 [36] or De-

scription Logic [8] ontologies. Coupled with the SLEEC rules, this offers the possibility of integrating advanced reasoning techniques for normative decision making. This can be done without an impact on the theoretical complexity stated in Proposition 5.

SLEEC-SLANG currently only supports untimed requirements, as the SLEEC rules conceived by [32] do. We plan to add timed deadlines to the rules in the future. We also plan to support more modeling choices, for example, the use of **otherwise** that is present in SLEEC DSL.

As described in Section 6, there exist works in the literature that support consistency-checking for SLEEC rules, as well as absence of redundancies, restrictive or insufficient rules. In the future, we plan to integrate these reasoning services in the elicitation framework of FRET, in order to automatically check requirements before generation of obligation inference engines or runtime monitors.

Data-Availability Statement

The code and all experimental data is publicly available as a VirtualBox VM at the following links:

- AMD architecture: <https://doi.org/10.5281/zenodo.17464288>
- ARM architecture: <https://doi.org/10.5281/zenodo.17465124>

The same material is also available on GitHub at the following address: <https://github.com/mirgit/FretishSLEEC>.

A Translation from SLEEC-DSL to SLEEC-SLANG

A SLEEC DSL rule can be transformed into a SLEEC-SLANG rule by rewriting its terms and variables. In this section, a rewriting function is proposed for that purpose. By applying τ to a SLEEC DSL rule, one can obtain the equivalent SLEEC-SLANG rule.

$$\begin{aligned}
 \tau(\text{ruleID } 'when' \text{ trigger } 'then' \text{ response}) &\rightarrow 'IF' \tau(\text{trigger}) 'THEN' \tau(\text{response}) \\
 \tau(\text{eventID } 'and' \text{ trigBoolExpr}) &\rightarrow \text{eventID}() ' \& ' \tau(\text{trigBoolExpr}) \\
 \tau(\text{trigBoolExpr } 'and' \text{ trigBoolExpr}) &\rightarrow \tau(\text{trigBoolExpr}) ' \& ' \tau(\text{trigBoolExpr}) \\
 \tau(\text{trigBoolExpr } 'or' \text{ trigBoolExpr}) &\rightarrow \tau(\text{trigBoolExpr}) ' | ' \tau(\text{trigBoolExpr}) \\
 \tau(\text{'not' trigBoolExpr}) &\rightarrow '!' \tau(\text{trigBoolExpr}) \\
 \tau(\{ 'numMeasureID' \} \text{ relOp value}) &\rightarrow \text{numMeasureID}(X_{id}) \& (X_{id} \tau(\text{relOp value})) \\
 \tau(\{ 'boolMeasureID' \}) &\rightarrow \text{boolMeasureID}() \\
 \tau(\text{eventID}) &\rightarrow \text{eventID}() \\
 \tau(\text{defeater defeaters}) &\rightarrow \tau(\text{defeater}) \tau(\text{defeaters}) \\
 \tau(\text{'unless' mBoolExpr}) &\rightarrow 'UNLESS' \tau(\text{mBoolExpr}) 'IWC' 'true' \\
 \tau(\text{'unless' mBoolExpr } 'then' \text{ response}) &\rightarrow 'UNLESS' \tau(\text{mBoolExpr}) 'IWC' \tau(\text{response}) \\
 \tau() &\rightarrow ''
 \end{aligned}$$

The translation above matches the order of the grammar rules in Table 1. Note that in this translation, X_{id} is a new variable name (var_name) for a numeric variable and we have omitted some of the standard steps for tokens like comparison.

B Full Datalog Obligation Inference Engine AQ-semantics

We present the automatically generated Datalog code for the obligation inference, AQ-semantics, in full and unaltered. Figure 1 presents a more compact but semantically equivalent program.

```
.decl See (X: symbol)
.decl Deaf (X: symbol)
.decl Blind (X: symbol)
.decl SayHi (X: symbol)
.decl Wave (X: symbol)
.decl TapShoulder (X: symbol)

SayHi ( X ) :- (See ( X )), !(Deaf ( X )) .
Wave ( X ) :- (See ( X )), (Deaf ( X )), !(Blind ( X )) .
TapShoulder ( X ) :- (See ( X )), (Deaf ( X )), (Blind ( X )) .

.input See
.input Deaf
.input Blind
.output SayHi
.output Wave
.output TapShoulder
```

Acknowledgments. The authors acknowledge the support of the MUR (Italy) Department of Excellence 2023–2027 for GSSI, of the PRIN project “HALO: etHical-aware AdjustabLe autOnomous systems” (grant 2022JKA4SL), of the PRIN PNRR 2022 project “RoboChor: Robot Choreography” (grant P2022RSW5W), and of the European HORIZON-KDT-JU research project MATISSE “Model-based engineering of Digital Twins for early verification and validation of Industrial Systems”, HORIZON-KDT-JU-2023-2-RIA , Proposal number: 101140216-2, KDT232RIA_00017.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases: The Logical Level. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edn. (1995)
2. Alfieri, C., Donati, D., Gozzano, S., Greco, L., Segala, M.: Ethical preferences in the digital world: The EXOSOUL questionnaire. In: 2nd International Conference on Hybrid Human-Artificial Intelligence. pp. 290–299 (2023). <https://doi.org/10.3233/FAIA230092>
3. Alfieri, C., Inverardi, P., Migliarini, P., Palmiero, M.: Exosoul: Ethical profiling in the digital world. In: Schlobach, S., Pérez-Ortiz, M., Tielman, M. (eds.) HHA1 2022: Augmenting Human Intellect - Proceedings of the First International Conference on Hybrid Human-Artificial Intelligence, Amsterdam, The Netherlands, 13-17 June 2022. Frontiers in Artificial Intelligence and Applications, vol. 354, pp. 128–142. IOS Press (2022). <https://doi.org/10.3233/FAIA220194>

4. Alidoosti, R., Lago, P., Poort, E., Razavian, M., Tang, A.: Incorporating ethical values into software architecture design practices. In: 19th International Conference on Software Architecture Companion. pp. 124–127 (2022). <https://doi.org/10.1109/ICSA-C54293.2022.00031>
5. Anderson, M., Anderson, S.L. (eds.): Machine Ethics. Cambridge University Press (2011)
6. Autili, M., De Sanctis, M., Inverardi, P., Pelliccione, P.: Engineering digital systems for humanity: A research roadmap. *ACM Trans. Softw. Eng. Methodol.* **34**(5) (May 2025). <https://doi.org/10.1145/3712006>, <https://doi.org/10.1145/3712006>
7. Awad, E., Dsouza, S., Kim, R., Schulz, J., Henrich, J., Shariff, A., Bonnefon, J.F., Rahwan, I.: The moral machine experiment. *Nature* **563**, 59–64 (2018). <https://doi.org/10.1038/s41586-018-0637-6>
8. Baader, F., Horrocks, I., Lutz, C., Sattler, U.: Introduction to Description Logic. Cambridge University Press (2017)
9. Commission, E., on Ethics in Science, E.G., Technologies, N., for Research, D.G., Innovation: Statement on artificial intelligence, robotics and 'autonomous' systems – Brussels, 9 March 2018. Publications Office (2018). <https://doi.org/doi/10.2777/531856>
10. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. *ACM Comput. Surv.* **33**(3), 374–425 (Sep 2001). <https://doi.org/10.1145/502807.502810>, <https://doi.org/10.1145/502807.502810>
11. Feng, N., Marsso, L., Yaman, S.G., Baatartogtokh, Y., Ayad, R., de Mello, V.O., Townsend, B.A., Standen, I., Stefanakos, I., Imrie, C., Rodrigues, G.N., Cavalcanti, A., Calinescu, R., Chechik, M.: Analyzing and debugging normative requirements via satisfiability checking. In: International Conference on Software Engineering. pp. 214:1–214:12. ACM (2024). <https://doi.org/10.1145/3597503.3639093>
12. Feng, N., Marsso, L., Yaman, S.G., Baatartogtokh, Y., Ayad, R., de Mello, V.O., Townsend, B.A., Standen, I., Stefanakos, I., Imrie, C., Rodrigues, G.N., Cavalcanti, A., Calinescu, R., Chechik, M.: SLEEC case studies (2024), <https://github.com/NickF0211/LEGOS-SLEEC>
13. Feng, N., Marsso, L., Yaman, S.G., Standen, I., Baatartogtokh, Y., Ayad, R., de Mello, V.O., Townsend, B.A., Bartels, H., Cavalcanti, A., Calinescu, R., Chechik, M.: Normative requirements operationalization with large language models. In: 32nd IEEE International Requirements Engineering Conference. pp. 129–141 (2024). <https://doi.org/10.1109/RE59067.2024.00022>
14. Feng, N., Marsso, L., Yaman, S.G., Townsend, B.A., Cavalcanti, A., Calinescu, R., Chechik, M.: Towards a formal framework for normative requirements elicitation. In: 38th IEEE/ACM International Conference on Automated Software Engineering. pp. 1776–1780 (2023). <https://doi.org/10.1109/ASE56229.2023.00152>
15. Future of Life Institute: Asilomar ai principles (2017), retrieved July, 2025, from <https://futureoflife.org/open-letter/ai-principles/>
16. Giannakopoulou, D., Pressburger, T., Mavridou, A., Rhein, J., Schumann, J., Shi, N.: Formal requirements elicitation with FRET. In: Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 26th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2020), Pisa, Italy, March 24, 2020. CEUR Workshop Proceedings, vol. 2584. CEUR-WS.org (2020), <https://ceur-ws.org/Vol-2584/PT-paper4.pdf>
17. Guarini, M.: Introduction: machine ethics and the ethics of building intelligent machines. *Topoi* **32**(2), 213–215 (2013)

18. Inverardi, P.: The european perspective on responsible computing. *Commun. ACM* **62**(4), 64 (2019). <https://doi.org/10.1145/3311783>
19. Jedlickova, A.: Ensuring ethical standards in the development of autonomous and intelligent systems. *IEEE Transactions on Artificial Intelligence* **5**(12), 5863–5872 (2024). <https://doi.org/10.1109/TAI.2024.3387403>
20. Katis, A., Mavridou, A., Pressburger, T., Schumann, J., Trinh, K.: FRET: Formal Requirements Elicitation Tool (2025), <https://github.com/NASA-SW-VnV/fret>
21. Kolyakov, K., Marsso, L., Feng, N., Quan, J., Chechik, M.: LEGOS-SLEEC: Tool for Formalizing and Analyzing Normative Requirements . In: 2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 33–36. IEEE Computer Society, Los Alamitos, CA, USA (May 2025). <https://doi.org/10.1109/ICSE-Companion66252.2025.00018>, <https://doi.ieeecomputersociety.org/10.1109/ICSE-Companion66252.2025.00018>
22. Madiega, T.A.: Artificial intelligence act (2024), [https://www.europarl.europa.eu/RegData/etudes/BRIE/2021/698792/EPRS_BRI\(2021\)698792_EN.pdf](https://www.europarl.europa.eu/RegData/etudes/BRIE/2021/698792/EPRS_BRI(2021)698792_EN.pdf), Think Tank European Parliament
23. Mirani, M., Raimondi, F., Troquard, N.: Towards efficient norm-aware robots' decision making using datalog (short paper). In: Proceedings of the 3rd Workshop on Bias, Ethical AI, Explainability and the role of Logic and Logic Programming co-located with the 23rd International Conference of the Italian Association for Artificial Intelligence (AIxIA 2024), Bolzano, Italy, November 26, 2024. CEUR Workshop Proceedings, vol. 3881, pp. 50–59. CEUR-WS.org (2024), <https://ceur-ws.org/Vol-3881/paper6.pdf>
24. OECD Council on Artificial Intelligence: Recommendation of the Council on Artificial Intelligence (2019), <https://legalinstruments.oecd.org/en/instruments/OECD-LEGAL-0449>, amended in 2024
25. Oviedo, J., Rodriguez, M., Trenta, A., Cannas, D., Natale, D., Piattini, M.: Iso/iec quality standards for ai engineering. *Computer Science Review* **54**, 100681 (2024). <https://doi.org/https://doi.org/10.1016/j.cosrev.2024.100681>, <https://www.sciencedirect.com/science/article/pii/S1574013724000650>
26. Russel, S.: Human Compatible: AI and the Problem of Control. Allen Lane, Bristol (2019)
27. Scholz, B. et al.: Souffl   (2025), <https://souffle-lang.github.io/index.html>, <https://souffle-lang.github.io/index.html>
28. Shahin, M., Hussain, W., Nurwidyantoro, A., Perera, H., Shams, R.A., Grundy, J.C., Whittle, J.: Operationalizing human values in software engineering: A survey. *IEEE Access* **10**, 75269–75295 (2022). <https://doi.org/10.1109/ACCESS.2022.3190975>, <https://doi.org/10.1109/ACCESS.2022.3190975>
29. Suri, S., Das, S.N., Singi, K., Dey, K., Sharma, V.S., Kaulgud, V.: Software engineering using autonomous agents: Are we there yet? In: 38th IEEE/ACM International Conference on Automated Software Engineering. pp. 1855–1857 (2023). <https://doi.org/10.1109/ASE56229.2023.00174>
30. Tolmeijer, S., Kneer, M., Sarasua, C., Christen, M., Bernstein, A.: Implementations in machine ethics: A survey. *ACM Computing Surveys* **53**(6), 1–38 (2020). <https://doi.org/10.1145/3419633>
31. Townsend, B., Parnell, K.J., Yaman, S.G., Nemirovsky, G., Calinescu, R.: Normative conflict resolution through human–autonomous agent interaction. *Journal of Responsible Technology* **21**, 100114 (2025). <https://doi.org/https://doi.org/10.1016/j.jrt.2025.100114>, <https://www.sciencedirect.com/science/article/pii/S2666659625000101>

32. Townsend, B.A., Paterson, C., Arvind, T.T., Nemirovsky, G., Calinescu, R., Cavalcanti, A., Habli, I., Thomas, A.: From pluralistic normative principles to autonomous-agent rules. *Minds Mach.* **32**(4), 683–715 (2022). <https://doi.org/10.1007/S11023-022-09614-W>
33. Troquard, N., De Sanctis, M., Inverardi, P., Pelliccione, P., Scoccia, G.L.: Social, legal, ethical, empathetic, and cultural rules: Compilation and reasoning. In: Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20–27, 2024, Vancouver, Canada. pp. 22385–22392. AAAI Press (2024). <https://doi.org/10.1609/AAAI.V38I20.30245>, <https://doi.org/10.1609/aaai.v38i20.30245>
34. UNESCO: Recommendation on the ethics of artificial intelligence, <https://www.unesco.org/en/artificial-intelligence/recommendation-ethics> (2022), <https://www.unesco.org/en/artificial-intelligence/recommendation-ethics>
35. United States Government: Blueprint for an AI Bill of Rights Making Automated Systems Work for the American People, <https://www.whitehouse.gov/wp-content/uploads/2022/10/Blueprint-for-an-AI-Bill-of-Rights.pdf> (2022), <https://www.whitehouse.gov/wp-content/uploads/2022/10/Blueprint-for-an-AI-Bill-of-Rights.pdf>
36. W3C OWL Working Group: Owl 2 web ontology language: Document overview. w3c recommendation (2009), available at <http://www.w3.org/TR/owl2-overview/>
37. Yaman, S.G., Burholt, C., Jones, M., Calinescu, R., Cavalcanti, A.: Specification and validation of normative rules for autonomous agents. In: Fundamental Approaches to Software Engineering - 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13991, pp. 241–248. Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_13, https://doi.org/10.1007/978-3-031-30826-0_13
38. Yaman, S.G., Ribeiro, P., Cavalcanti, A., Calinescu, R., Paterson, C., Townsend, B.A.: Specification, validation and verification of social, legal, ethical, empathetic and cultural requirements for autonomous agents. *Journal of Systems and Software* **220**, 112229 (2025). <https://doi.org/10.1016/J.JSS.2024.112229>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Deciding Serializability in Network Systems

Guy Amir¹, Mark Barbone¹, Nicolas Amat², and Jules Jacobs^{1,3}

¹ Cornell University, Ithaca, USA

{gda42, mlb494, jj758}@cornell.edu

² DTIS, ONERA, Université de Toulouse, Toulouse, France

nicolas.amat@onera.fr

³ Jane Street Capital, New York City, USA

Abstract. We present the SER modeling language for automatically verifying *serializability* of concurrent programs, i.e., whether every concurrent execution of the program is equivalent to some serial execution. SER programs are suitably restricted to make this problem decidable, while still allowing for an *unbounded* number of concurrent threads of execution, each potentially running for an *unbounded* number of steps. Building on prior theoretical results, we give the first automated end-to-end decision procedure that either proves serializability by producing a checkable certificate, or refutes it by producing a counterexample trace. We also present a network-system abstraction to which SER programs compile. Our decision procedure then reduces serializability in this setting to a Petri net reachability query. Furthermore, in order to scale, we curtail the search space via multiple optimizations, including Petri net slicing, semilinear-set compression, and Presburger-formula manipulation. We extensively evaluate our framework and show that, despite the theoretical hardness of the problem, it can successfully handle various models of real-world programs, including stateful firewalls, BGP routers, and more.

1 Introduction

In the domain of concurrent systems, from databases to software-defined networks (SDNs) [62, 100], a cornerstone correctness criterion is *serializability*: every concurrent execution must produce outcomes equivalent to some serial ordering of requests. Violations of serializability can lead to subtle anomalies, such as lost updates in databases or routing cycles in SDNs. While we can check serializability for a fixed number of requests with known execution traces (e.g., by enumerating all possible interleavings), the problem is undecidable for general programs, requiring techniques such as runtime verification or incomplete bounded model checking [5, 16, 45, 49, 79, 86, 87, 97, 98].

However, Bouajjani et al. [18] have shown (as a special case of bounded-barrier linearizability) that for programs with bounded-size state, this problem is decidable even for an *unbounded* number of in-flight threads, each performing an *unbounded* number of steps. The purpose of this paper is to make this theoretical decidability result a reality by designing the first decision procedure

and putting forth practical algorithms that either prove serializability (with a proof certificate) or prove non-serializability (with a counterexample trace). We illustrate the problem by example:

```

1  // request handler
2  request main:
3      X := 1 // X is global
4      y := X // y is local
5      X := 0
6      return y

```

Listing 1.1: Without yielding (serializable)

```

1  request main:
2      X := 1
3      yield // another request
4      y := X // can read 0!
5      X := 0
6      return y

```

Listing 1.2: With yielding (not serializable)

```

1  request main:
2      // lock
3      while (L == 1):
4          yield
5          L := 1
6
7      X := 1
8      yield
9      y := X
10     X := 0
11
12     // unlock
13     L := 0
14     return y

```

Listing 1.3:
With yielding and a spin-
lock (serializable)

These examples are written in our modeling language called SER. A SER program has a set of named **request handlers** (one handler, **main**, in the examples) that are arbitrarily invoked concurrently by the external environment. Each incoming request processes its request handler's body until it returns a value as its **response**. Concurrency is managed by the **yield** statement, which pauses the current request and gives other requests a chance to run. SER programs have uppercase **global shared variables** (**X** in the examples) and lowercase **request-local variables** (**y** in the examples). The first program (Listing 1.1) is clearly serializable because there are no yields, and hence, no interleavings: each **main** request returns 1. In the second program (Listing 1.2), the **yield** allows interleavings that make the program *non-serializable*. For instance, consider two concurrent requests to **main**: Request A executes [**X** := 1] then yields to Request B; which then executes [**X** := 1], yields to itself, reads **X** (getting 1), sets [**X** := 0], and returns 1. Finally, Request A resumes, reads **X** (now 0), and returns 0. This produces the multiset {(**main**, 0), (**main**, 1)} of (request, response) pairs, which is impossible in any serial execution (where all **main** requests return 1 and never 0). Of course, having yields does not guarantee that an execution is necessarily not serializable, as observed in the third snippet (Listing 1.3). This program uses an additional lock variable (**L**), which guarantees that even if an interleaving occurs, the program is semantically equivalent to the first one. These examples demonstrate that reasoning about serializability can be complex even for very simple programs with few requests running concurrently. For a tour of additional examples, we refer the reader to Appendix A of our extended paper [9].

Problem Definition. Formally, we define the **observable execution** of a SER program as a multiset of (request, response) pairs induced by a specific interleaving. The **observable behavior** of a SER program is the set of all possible observable executions that can occur such that the requests are executed concurrently to obtain their paired responses. A SER program is **serializable** if every observable behavior is achievable serially (without interleavings). Differently put, removing all **yield** statements does not change the program’s semantics. *This paper aims to present the SER language and decision procedure for this problem.* In particular, SER is the first toolchain to **automatically** prove serializability without requiring manual work by the user.

Challenges. To our knowledge, no prior implementation exists that can automatically generate proof certificates for this class of concurrent systems. Why not? Our decision procedure builds on Bouajjani et al.’s reduction from serializability to Petri net (PN) reachability [18]. However, since PN reachability is **Ackermann**-complete [34,66], a naive implementation would fail on all but simple programs.

Our Approach. To address this, we first introduce the abstraction of *network systems* (NS) — modeling concurrent programs where users send *requests* that manipulate local and shared state before returning *responses*. A SER program is compiled into a network system, on which our decision procedure operates via reduction to Petri net reachability and semilinear set analysis. We note that while our approach is sound (never incorrectly claims serializability), the underlying reachability query may time out on complex instances, limiting completeness in practice (this is unavoidable, given the **Ackermann**-hardness of the problem). Towards this end, we developed multiple optimizations to make the approach practical, including Petri net slicing, semilinear set compression, and additional manipulations with Presburger formulas. As we demonstrate, these optimizations reduce the search space by *orders of magnitude*, enabling us to scale to non-trivial programs. Finally, we extensively evaluated our SER toolchain on various programs, covering a broad spectrum of features such as loops, branching, locks, and nondeterminism; as well as SDN-inspired examples such as stateful firewalls, BGP routers, and more. To our knowledge, this leads to the first *implemented* decision procedure that: (i) automatically *proves* serializability for *unbounded* executions; (ii) generates *proof certificates*; and (iii) handles *non-trivial programs*.

Contributions. We introduce in §2 our SER language and the Network System program abstraction that captures the essence of concurrent systems. In §3 we present the core decision procedure with proof certificates, and our various optimizations. The implementation of the SER toolchain is covered in §4, and its evaluation is presented in §5. We discuss related work in §6 and conclude in §7. Our tool, benchmarks, and experiments are available as an accompanying artifact [3].

2 Problem Definition

2.1 Background

Petri nets. A Petri net is $N = (P, T, \text{pre}, \text{post}, M_0)$ with a set of places P , a set of transitions T , flow functions $\text{pre}, \text{post} : T \rightarrow \mathbb{N}^P$, and an initial marking (token distribution) $M_0 \in \mathbb{N}^P$. A transition t is *enabled* at marking $M \in \mathbb{N}^P$ if $M \geq \text{pre}(t)$ coordinate-wise, i.e., M provides at least as many tokens as required by $\text{pre}(t)$. If an enabled transition t *fires*, it produces the marking M' (denoted $M \xrightarrow{t} M'$), with $M' = M - \text{pre}(t) + \text{post}(t)$ consuming input tokens and producing output tokens. This can extend naturally to a sequence of firings $\sigma = t_1 \cdots t_k$ (denoted $M \xrightarrow{\sigma} M'$), giving rise to a sequence of markings $M_0, \dots, M_k$ with $M = M_0$, $M' = M_k$, and $M_i \xrightarrow{t_i} M_{i+1}$ for all i . We define the set $R(N) = \{M \mid \exists \sigma \in T^*. M_0 \xrightarrow{\sigma} M\}$ to include all markings reachable from the initial state M_0 . The *reachability problem* asks, given a Petri net N and a marking M , whether $M \in R(N)$. Specifically, we focus on reachability of a linear-constraint formula $\mathcal{F}$; it is **SAT** if some marking $M \in R(N)$ satisfies $\mathcal{F}$ (denoted $M \models \mathcal{F}$), and otherwise **UNSAT** (see the toy example in Appendix B of our extended paper [9]). Surprisingly, even the *unbounded* case, where places hold arbitrarily many tokens, is decidable [60, 64, 73], although **Ackermann**-complete [34, 66].

Verdict proofs. If $\mathcal{F}$ is reachable, a witness sequence $\sigma \in T^*$ with $M_0 \xrightarrow{\sigma} M$ and $M \models \mathcal{F}$ serves as a proof, and is verifiable by simulation of the Petri net. If $\mathcal{F}$ is unreachable, there exists [67] an inductive Presburger certificate C proving non-reachability: (i) $M_0 \models C$, (ii) $\forall t \in T \quad (M \models C \wedge M \xrightarrow{t} M') \Rightarrow M' \models C$, and (iii) $C \Rightarrow \neg \mathcal{F}$.

Semilinear sets and Parikh's theorem. A set $S \subseteq \mathbb{N}^k$ is *semilinear* if $S = \bigcup_{i=1}^m \{\mathbf{b}_i + \sum_{j=1}^{r_i} n_j \mathbf{p}_{i,j} \mid n_j \in \mathbb{N}\}$ for some $\mathbf{b}_i, \mathbf{p}_{i,j} \in \mathbb{N}^k$. Such sets coincide with those definable by *Presburger arithmetic* [83]. By Parikh's theorem [80], the *Parikh image* of any context-free language (CFL) is semilinear. For an alphabet $\Sigma = \{a_1, \dots, a_k\}$ and a word $w \in \Sigma^*$, the *Parikh image* of w is the vector $\text{Parikh}(w) = (|w|_{a_1}, \dots, |w|_{a_k}) \in \mathbb{N}^k$, where $|w|_{a_i}$ denotes the number of occurrences of the symbol a_i in w . For a language $L \subseteq \Sigma^*$, we write $\text{Parikh}(L) = \{\text{Parikh}(w) \mid w \in L\} \subseteq \mathbb{N}^k$.

Deciding serializability in unbounded systems. Bouajjani et al. [18] have proved that serializability in unbounded systems reduces to Petri net reachability, as a special case of *bounded-barrier linearizability*.

2.2 The SER Language

Our SER syntax is defined as follows:

Expression	$e ::= 0 \mid 1 \mid 2 \mid \dots$	numeric constant
	$\mid ?$	nondeterministic value (0/1)
	$\mid x := e \mid x$	write/read local variable
	$\mid X := e \mid X$	write/read global variable
	$\mid e_1 == e_2$	equality test
	$\mid e_1; e_2$	sequencing
	$\mid \text{if}(e_1)\{e_2\}\text{else}\{e_3\}$	conditional
	$\mid \text{while}(e_1)\{e_2\}$	while loop
	$\mid \text{yield}$	yield to scheduler

Program $P_0 ::= \text{request } name_1\{e_1\} \dots \text{request } name_n\{e_n\}$ set of handlers

Given a program P_0 , we write $name_i\{e_i\} \in P_0$ for each of the handlers $name_i\{e_i\}$. Our semantics is standard and fully formalized in Appendix C of our extended paper [9]. In addition, arithmetic extensions are supported in the tool [3] but omitted here for brevity.

2.3 Network System

We now present our abstract network system (NS) model, motivated by software-defined networks. In the networking domain, spawning a request corresponds to sending a *packet*, with each local variable mapped to a unique *packet header field*; global variables correspond to variables on *programmable switches*, as they are shared among all requests visiting the switch. Throughout this paper, we use the term *request* to refer to a concurrent computation unit. We define a network system $\mathcal{N}$ as a tuple $(G, L, REQ, RESP, g_0, \delta, req, resp)$ where:

- G is a set of *global network states* (e.g., the values of variables on a switch)
- L is a set of *local packet states* (e.g., packet header values)
- REQ is a finite set of *request labels* (each marked $\blacklozenge_{\text{req}}$)
- $RESP$ is a finite set of *response labels* (each marked $\blacklozenge_{\text{resp}}$)
- $g_0 \in G$ is the *initial global state* of the network system
- $req \subseteq REQ \times L$ maps each request to its corresponding (initial) local state
 - this represents externally spawning a packet matching the request type
- $resp \subseteq L \times RESP$ maps a (final) local state to its corresponding response
 - this represents a packet exiting the network and returning the computation
- $\delta \subseteq (L \times G) \times (L \times G)$ defines atomic execution steps that update both global and local state (this represents a packet doing a single hop in the network)

Request and response. A *request* label $\blacklozenge_{\text{req}} \in REQ$ spawns a request (i.e., a packet/thread) on which a concurrent computation is executed; a *response* label $\blacklozenge_{\text{resp}} \in RESP$ is its returned value. The pair $(\blacklozenge_{\text{req}}, \blacklozenge_{\text{resp}})$ captures the input/output behavior of a single request from a single concurrent execution of the NS.

States. A *network state* is a triple $(g, \mathcal{R}, \mathcal{Z})$ where $g \in G$ is the global network state, $\mathcal{R} \in \text{Multiset}(L \times REQ)$ is a multiset of in-flight requests (i.e., local assignments of each thread in the current timestep, and the original request label that spawned it), and $\mathcal{Z} \in \text{Multiset}(REQ \times RESP)$ is a multiset of completed request/response pairs. We write $\uplus$ for multiset union. The initial global state is $(g_0, \emptyset, \emptyset)$.

Transition rules. A transition $\longrightarrow$ either (1) spawns a request; (2) advances one request via δ ; or (3) returns a response. When no further steps remain, $\mathcal{Z}$ is the final multiset of request/response pairs that arose during the NS run.

$$\begin{aligned}
& \text{(New Request)} \quad \frac{(\blacklozenge_{\text{req}}, \ell) \in req}{(g, \mathcal{R}, \mathcal{Z}) \rightarrow (g, \mathcal{R} \uplus \{(\ell, \blacklozenge_{\text{req}})\}, \mathcal{Z})} \\
& \text{(Processing Step)} \quad \frac{((\ell, g), (\ell', g')) \in \delta}{(g, \mathcal{R} \uplus \{(\ell, \blacklozenge_{\text{req}})\}, \mathcal{Z}) \rightarrow (g', \mathcal{R} \uplus \{(\ell', \blacklozenge_{\text{req}})\}, \mathcal{Z})} \\
& \text{(Response)} \quad \frac{(\ell, \blacklozenge_{\text{resp}}) \in resp}{(g, \mathcal{R} \uplus \{(\ell, \blacklozenge_{\text{req}})\}, \mathcal{Z}) \rightarrow (g, \mathcal{R}, \mathcal{Z} \uplus \{(\blacklozenge_{\text{req}}, \blacklozenge_{\text{resp}})\})}
\end{aligned}$$

Serializability. An *interleaved run* is a complete execution $(g_0, \emptyset, \emptyset) \rightarrow^* (g_n, \emptyset, \mathcal{Z})$:

$$(g_0, \emptyset, \emptyset) \rightarrow (g_1, \mathcal{R}_1, \mathcal{Z}_1) \rightarrow \cdots \rightarrow (g_n, \mathcal{R}_n, \mathcal{Z}_n), \quad \text{with } \mathcal{R}_n = \emptyset, \mathcal{Z}_n = \mathcal{Z}.$$

It is *serial* if each $\mathcal{R}_i$ has *at most* one request. Intuitively, serial runs have at most one in-flight request at a time. Given NS $\mathcal{S}$, let $\text{Int}(\mathcal{S})$ and $\text{Ser}(\mathcal{S})$ respectively denote the (infinite) sets of request/response multisets, for interleaved and serial runs of $\mathcal{S}$:

$$\text{Int}(\mathcal{S}) = \{ \mathcal{Z} \mid \exists \text{ interleaved run } (g_0, \emptyset, \emptyset) \rightarrow^* (g_n, \emptyset, \mathcal{Z}) \},$$

$$\text{Ser}(\mathcal{S}) = \{ \mathcal{Z} \mid \exists \text{ serial run } (g_0, \emptyset, \emptyset) \rightarrow^* (g_n, \emptyset, \mathcal{Z}) \} \subseteq \text{Int}(\mathcal{S}).$$

An NS $\mathcal{S}$ is *serializable* if $\text{Int}(\mathcal{S}) = \text{Ser}(\mathcal{S})$, i.e., every multiset of request/response pairs obtained by an interleaved execution can also be obtained serially.

2.4 Translating SER Programs to Network Systems

The NS abstraction not only captures concurrent behaviors in software-defined networks but also enables a natural translation from SER programs. Given a SER program P_0 with local variables (**vars**), global variables (**VARS**), and mappings ρ, g from these to a finite value set $\mathbb{V}$, we define the initial local/global states ρ_0

and g_0 assigning 0 to all local and global variables, respectively. Using the small-step semantics ($\Rightarrow$, defined in full in Appendix C of our extended paper [9]), we construct the NS $(G, L, REQ, RESP, g_0, \delta, req, resp)$:

$$\begin{aligned}
G &= \{g : \mathbf{VARS} \rightarrow \mathbf{V}\}, \\
L &= \{(e, \rho) \mid \rho : \mathbf{vars} \rightarrow \mathbf{V}, \exists name_i\{e_i\} \in P_0 \text{ s.t.} \\
&\quad e = e_i \text{ or } e \text{ is a suffix of } e_i \text{ starting after a } \mathbf{yield} \text{ statement}\}, \\
REQ &= \{name_i \mid name_i\{e_i\} \in P_0\}, \quad RESP = \mathbf{V}, \\
req &= \{(r, \ell) \mid r = name_i \in REQ, name_i\{e_i\} \in P_0, \ell = (e_i, \rho_0) \in L\}, \\
resp &= \{(\ell', r') \mid \exists v \in \mathbf{V}. \ell' = (v, \rho') \in L, r' = v \in RESP\}, \\
\delta &= \{((e, \rho), g) \rightarrow ((e', \rho'), g') \mid (e, \rho), (e', \rho') \in L, g, g' \in G, \langle e, \rho, g \rangle \Rightarrow \langle e', \rho', g' \rangle\}.
\end{aligned}$$

Example. We construct the NS for the non-serializable example in Listing 1.2:

- The set G is defined as $G = \{[X=0], [X=1]\}$.
- The initial global state is defined as $g_0 = [X=0]$.
- The set L is defined as all reachable local states, i.e., pairs of assignments (such as $[y=0], [y=1]$) coupled with all reachable SER programs (continuations of a program at a point of execution). For example, the reachable programs for Listing 1.2 are depicted as code snippets in Fig. 1.
- The set of requests is $REQ = \{\blacklozenge_{\text{main}}\}$.
- The set of responses is $RESP = \{\blacklozenge_0, \blacklozenge_1\}$.
- The function δ is presented in Fig. 10 (Appendix D of [9]).

We depict in Fig. 1 the explicit network system that serves as a mapping from requests ($\blacklozenge_{\text{main}}$) to responses ($\blacklozenge_0, \blacklozenge_1$). We note that, for simplicity, we depict only *reachable* states.

3 Formal Results

3.1 The Algorithm (without Optimizations)

Given a network system $\mathcal{S} = (G, L, REQ, RESP, g_0, \delta, req, resp)$ we run the following steps:

Step 1: Serializability automaton. We define an NFA $\mathcal{A}_{\text{ser}}(\mathcal{S}) = (Q, \Sigma, \delta^A, q_0, F)$, with $Q = G$, $F = G$, $q_0 = g_0$, over an alphabet $\Sigma = \{(\blacklozenge_{req}/\blacklozenge_{resp}) \mid \blacklozenge_{req} \in REQ, \blacklozenge_{resp} \in RESP\}$. We let each transition correspond to a request/response pair: $\delta^A \subseteq Q \times \Sigma \times Q$, $q \xrightarrow{\blacklozenge_{req}/\blacklozenge_{resp}} q'$, iff $\mathcal{S}$ is in global state q and issues a request $\blacklozenge_{req}$, then upon some *full serial execution* it eventually transitions to

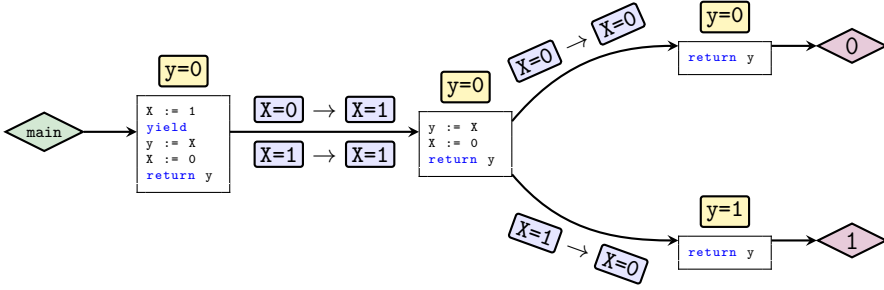


Fig. 1: The network system for Listing 1.2. Local states show the variable assignments (yellow rectangles ) and the remaining code; edges indicate transitions of global states (blue rectangles ). Requests and responses appear as  (green) and  (red) diamonds, respectively. From left to right: _{main} spawns a request with $[y=0]$ and the full program; after yielding, δ steps with global state $[X=1]$ and local state $[y=0]$, then updates y based on the global value, returning it as the final response (either ₀ or ₁).

global state q' and returns response _{resp}. Its language $L(\mathcal{A}_{\text{ser}}(\mathcal{S})) \subseteq \Sigma^*$ is exactly the set of serial request/response traces. Hence, by definition, it holds that applying the Parikh image gives the set of all multisets of request/response pairs obtained by serial executions: $\text{Ser}(\mathcal{S}) = \text{Parikh}(L(\mathcal{A}_{\text{ser}}(\mathcal{S}))) \subseteq \mathbb{N}^\Sigma$.

Example. For Listing 1.2, the NS in Fig. 1 gives rise to the Serial NFA in Fig. 2. A trace of request/response pairs is accepted by the NFA iff some serial execution of the program induces it. Here, serial runs produce only (_{main}/₁), and the only reachable global state is $[X=0]$.

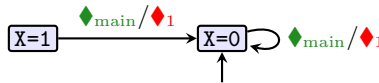


Fig. 2: Serial NFA of Listing 1.2.

Step 2: Interleaving Petri net. Next, we translate the NS into a Petri net $N_{\text{int}}(\mathcal{S})$. The *non-sink places* of the PN represent either (i) global state assignments, or (ii) local states of in-flight packets. The *sink places* represent request/response pairs of terminated packets. We define the *transitions* between states to correspond to the δ , *req*, and *resp* mappings of the NS (the *req* transitions can fire without any input tokens in order to correspond to initializing arbitrarily many requests externally). Finally, we define the initial marking M_0

to be a single token in the place corresponding to the initial global state g_0 . This construction (which is fully formalized in Appendix E of [9]) guarantees that any reachable marking M (i.e., $M_0 \rightarrow^* M$), when projected (π) to the sink places, corresponds to a multiset of ($\blacklozenge_{req}/\blacklozenge_{resp}$) pairs of the NS, as obtained by an interleaving (and vice versa), i.e., $\text{Int}(\mathcal{S}) = \{\pi(M) \mid M \in R(N_{\text{int}}(\mathcal{S}))\}$.

Example. In our running example, the NS gives rise to the PN in Fig. 3, encoding all possible interleavings. The places P_2 and P_3 represent the global states $[X=1]$ and $[X=0]$, respectively, while the places P_1, P_4, P_5 , and P_6 capture the local states of in-flight requests, i.e., the remaining program code coupled with the assignments to each request’s local variables. Similarly, places P_7 and P_8 respectively correspond to responses $\blacklozenge_1$ and $\blacklozenge_0$. Each token either models an active request, a completed request/response pair, or — when residing in a global-state place — the current global state of the NS. Finally, transitions implement the network system’s mappings ($\delta/req/resp$): they either spawn a new request (e.g., transition t_1 , producing $\blacklozenge_{\text{main}}$ based on req), advance the program by one step (e.g., t_2, t_3, t_4 , and t_5 , based on δ), or return a response (e.g., transitions t_6 and t_7 , based on $resp$).

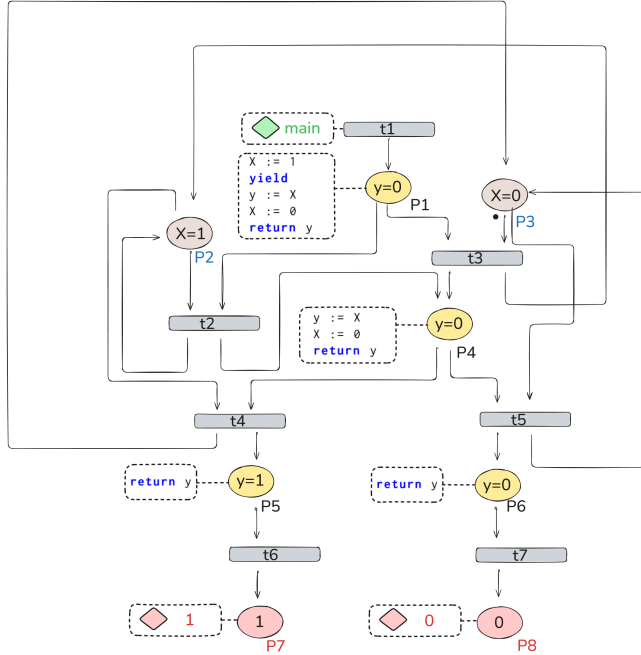


Fig. 3: The PN encoding interleaved executions of the program in Listing 1.2.

Step 3: Non-serializable set. Let $\text{NonSer}(\mathcal{S}) = \mathbb{N}^{|\Sigma|} \setminus \text{Ser}(\mathcal{S})$, i.e., all multisets of ($\blacklozenge_{\text{req}}$ / $\blacklozenge_{\text{resp}}$) pairs that *cannot* be obtained via a serial execution of NS $\mathcal{S}$.

Example. Regarding the aforementioned program, we automatically generate the following reachability query^a for the Petri net in Fig. 3, encoding a target semilinear set by imposing the following constraints on the token distribution:

$$\mathcal{F} : \quad P_1 = 0 \wedge P_2 \geq 0 \wedge P_3 \geq 0 \wedge P_4 = 0 \wedge P_5 = 0 \wedge P_6 = 0 \wedge P_7 \geq 0 \wedge P_8 \geq 1.$$

This set requires no tokens on P_1, P_4, P_5, P_6 , at least one token on P_8 (i.e., a response $\blacklozenge_0$), and any number of tokens on P_2, P_3, P_7 .

^a If not for the equality constraints, the problem would have been considered a Petri net *coverability* query, which is easier [84].

Step 4: Decision & validation. We ask whether there *exists* a reachable marking M of $N_{\text{int}}(\mathcal{S})$ such that $M \models \mathcal{F}$. As $\mathcal{F}$ encodes $\text{NonSer}(\mathcal{S})$, this is equivalent to a marking M such that $M_0 \rightarrow^* M$ and

$$\pi(M) \in \text{Int}(\mathcal{S}) \quad \wedge \quad \pi(M) \in \text{NonSer}(\mathcal{S}).$$

SAT : yields a counterexample interleaving M with $\pi(M) \notin \text{Ser}(\mathcal{S})$, validated by simulation of the network system $\mathcal{S}$.

UNSAT : yields an inductive invariant of $N_{\text{int}}(\mathcal{S})$, back-translated to an NS-level proof of serializability (see an example in Appendix F of our extended paper [9]).

Example. In our running example, the target semilinear set $\mathcal{F}$ is, in fact, reachable. For instance, it includes the following marking:

$$M^* = \{P_3(1), P_7(1), P_8(1)\}$$

which is reachable by the PN in Fig. 3. The full firing sequence leading to marking M^* is given in Table 3 (in Appendix G of our extended paper [9]). Specifically, this reachable marking encodes the outputs $\{\blacklozenge_{\text{main}}/\blacklozenge_0, \blacklozenge_{\text{main}}/\blacklozenge_1\}$ which, indeed, can only be induced by a non-serial execution of Listing 1.2.

Complexity analysis. The core algorithm reduces serializability checking to Petri net reachability with target semilinear sets. Since the serial executions form a regular language (step 1), their Parikh image is effectively semilinear by Parikh’s theorem, with size exponential in the NFA. The interleaving Petri net (step 2) has $O(|G| + (|REQ| \times |L|) + (|REQ| \times |RESP|))$ places and $O(|REQ| \times (1 + |\delta| + |RESP|))$ transitions. The reachability query (step 3) asks whether

the Petri net can reach the complement of a semilinear set, which is decidable but Ackermann-complete [34, 66]. Without optimizations, even simple examples can generate Petri nets with hundreds of places and exponentially-sized semilinear constraints, making the approach impractical. Our optimizations (see subsec. 3.2) *drastically* reduce both the Petri net size and the semilinear set complexity, as we elaborate next.

3.2 Optimizations

We apply four optimizations to the base algorithm to control intermediate blow-up in the size of both the PN and the constructed semilinear set. An extensive empirical evaluation of these optimizations appears in Appendix I of our extended paper [9].

(1) Bidirectional slicing. When solving Petri net reachability, many places and transitions might be irrelevant to the specific target set [85]. We slice them before symbolic reasoning by combining forward and backward passes: the forward pass over-approximates the places reachable from M_0 ; and symmetrically, the backward pass traverses in reverse from any place that can influence a target constraint (hence over-approximating the places that can contribute to it). We iteratively remove non-forward-reachable and non-backward-relevant places and transitions, to a fixed point. Appendix H of our extended paper [9] illustrates this (Fig. 14) and proves soundness (Theorem 1):

Theorem 1 (Bidirectional Slicing Soundness). *Let $N = (P, T, \text{pre}, \text{post}, M_0)$ be a Petri net and S a target set. Let $N' = (P', T', \text{pre}|_{P' \times T'}, \text{post}|_{P' \times T'}, M_0|_{P'})$ be the sliced net. Then S is reachable from N iff it is reachable from N' .*

(2) Semilinear set pruning. A semilinear set $S = \bigcup_{i=1}^m L_i$ with $L_i = \{\mathbf{b}_i + \sum_{\mathbf{p} \in P_i} n_p \mathbf{p} \mid n_p \in \mathbb{N}\}$ may contain redundant period vectors or components. Thus, during construction, we: (1) remove any period vector $\mathbf{p} \in P_i$ expressible as a nonnegative combination of $P_i \setminus \{\mathbf{p}\}$; and (2) drop L_i when $L_i \subseteq L_j$ (for $i \neq j$). This pruning keeps formulas compact and solver calls tractable.

(3) Generating fewer constraints. When computing the Parikh image of a regular expression as a semilinear set, most regex operations can be implemented without an exponential blow-up. However, the Kleene star is a notable exception. Given $S = \bigcup_{i=1}^m L_i$, the Kleene closure S^* can be expressed as a semilinear set by:

$$S^* = \bigcup_{I \subseteq \{1, \dots, m\}} \left\{ \sum_{i \in I} \mathbf{b}_i + \sum_{\mathbf{p} \in \bigcup_{i \in I} (P_i \cup \{\mathbf{b}_i\})} n_p \mathbf{p} \right\},$$

yielding 2^m components. To mitigate this: (i) if $L_i = \{\mathbf{b}_i\}$ (period-less component), factor it out, star the rest, then add $\mathbf{b}_i$ as a period; (ii) if $L_i = \{\sum_{\mathbf{p} \in P_i} n_p \mathbf{p}\}$ (zero base), likewise star the rest and add each $\mathbf{p} \in P_i$ as a period vector to the resulting set. Each such case halves the component count and circumvents exponential blow-ups.

(4) Strategic Kleene elimination order. We use *Kleene’s algorithm* [58] to translate the serializability NFA into a regex. The size of the generated semilinear set is not only impacted by how the semilinear set operations are implemented, but also by what *specific* regular expression is given as input: a single regular language may be represented by a number of equivalent regexes, each of different complexity. In particular, as Kleene star can cause a large blow-up in the semilinear set size, we are especially sensitive to the *star height* of the generated regex. Naive Kleene elimination may introduce many nested stars. We reduce this by strategically choosing to eliminate lower-degree states first:

$$q^* = \arg \min_{q \in Q} (|\delta_{\text{in}}^A(q)| + |\delta_{\text{out}}^A(q)|).$$

As we demonstrate in Appendix I of [9], our optimizations expedite the search procedure and make the representations *significantly* more compact. This, in turn, enables deciding serializability for instances that are otherwise intractable.

4 Implementation

4.1 Code Architecture

We implemented our approach in SER [3], a publicly available toolchain written mostly in Rust. SER implements an end-to-end serializability checker for a given input program. If the program is serializable, we return a proof thereof; otherwise, if it is not serializable, a counterexample is given to the user for an interleaving that can result in request/response pairs that are unattainable in any serial execution. Our workflow translates the decidability problem to an equivalent Petri net reachability question (for an unbounded number of tokens), in which (i) the Petri net represents all possible interleavings of the program; and (ii) the reachability query represents a semilinear set (equivalently, a Presburger arithmetic encoding) of all request/response pairs that *cannot* be obtained by any serial execution. As Petri net reachability is **Ackermann**-complete [34, 66], we added various optimizations to expedite the search process, both at the PN level and the property-encoding level. The pipeline of SER is depicted in Fig. 4, and includes:

1. **Input & parsing.** Our framework receives either a SER program with the syntax described in §2, or a JSON file directly encoding a network system. In the case of the former, an additional step takes place, parsing the input to an expression tree that is translated to the equivalent NS.
2. **Petri net conversion.** The NS is then translated into a Petri net which represents all possible interleavings. The PN is encoded in the de facto standard NET format, to support off-the-shelf PN model checkers.
3. **Semilinear conversion.** We generate a semilinear set encoding all non-serializable outputs, via translation of the serialized NFA (e.g., Fig. 2) to a regex, which is then projected (via the Parikh image) and complemented. At the end of the pipeline, an XML-formatted output encodes a reachability query that encapsulates constraints over the PN token count.

4. **Reachability engine.** The PN and the reachability query are fed to a PN model checker, which combines *bounded model checking* (BMC) [14] in search of a counterexample; and *state equation reasoning* [74] in order to prove non-reachability. In order to expedite the search, “large” (PN, query) pairs are replaced with multiple sliced PNs (generated by the reachability engine), each coupled with a sub-query encoding a separate disjunct. The disjuncts are solved on the fly, until reaching SAT, in which case, we have a counterexample; otherwise, if all disjuncts are UNSAT, we render the original program as serializable.
5. **Proof & certification.** If SAT, we reconstruct and validate an NS-level counterexample. Otherwise, if all disjuncts are UNSAT, we extract per-disjunct proofs and “stitch” these to a single inductive serializability certificate, which we then project to the NS and validate (i) initiation, (ii) inductiveness, and (iii) query refutation.
6. **Instrumentation & logging.** Throughout the pipeline, we record various intermediate representations and performance metrics.

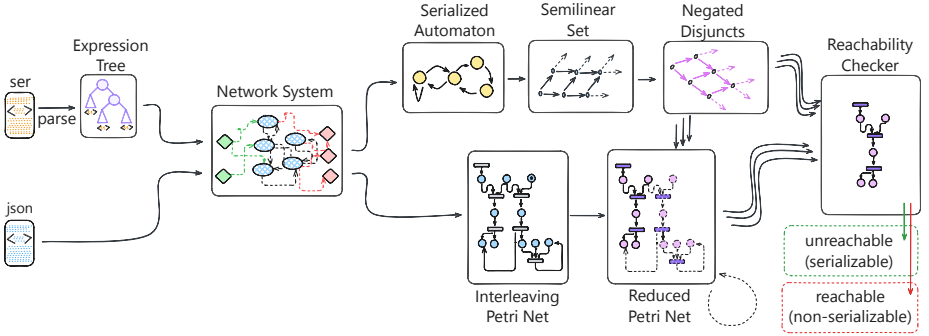


Fig. 4: Full program flow (simplified, without backward arrows to the NS level).

4.2 Benchmark Overview

To the best of our knowledge, ours is the first and only tool to: (i) statically check serializability on *unbounded* programs; and (ii) *prove serializability holds*. Thus, due to a lack of standard benchmarks for evaluating serializability, we assembled a suite of dozens of benchmarks (as part of our accompanying artifact [3]). We believe this is the first benchmark suite for serializability in this setting, aiming to connect SER programs to practical, real-world analogues. These include both serializable and non-serializable instances encoded in both SER and JSON formats, and covering a broad range of features, including arithmetic, locks, loops, non-determinism, and more (see an overview of all our benchmarks in Table 4 of Appendix I in our extended paper [9]). We note that although the benchmarks

themselves are not the main part of the paper, we believe that they have merit on their own, due to their relevance to various real-world systems of interest. Specifically, we wish to note our suite of benchmarks encoding *network & system protocols* (see Table 2), which include models of stateful firewalls, BGP routing programs, network monitors, and more — as motivated by real-world concurrency problems in this domain. One such example is our *routing-cycle benchmark* in software-defined networks (motivated by [76]). Another example is our *snapshot isolation benchmark* (see Appendix A of our extended paper [9]), which was motivated by a real database bug, namely, duplicate-key errors [30] in the popular *CockroachDB* system [2]. In both cases and others, the non-serializable behavior was automatically identified by our toolchain.

5 Evaluation

Experimental setup. All experiments were run on a Lenovo ThinkPad P16s, with 16 AMD CPU cores and 64 GB of RAM, running Ubuntu 24.04.2. We use *SMPT* [7] (built upon *Z3* [35]) as our backend Petri net model checker. Our code and benchmarks are publicly available [3].

Results. We ran *SER* on all 47 benchmarks, out of which 27 are serializable, and the remaining 20 are non-serializable. For each benchmark, we measured the time for deciding the reachability query, as well as the overall time, including validation of the invariant proof (if serializable) or of the counterexample (if not serializable). These experiments ran in parallel on 16 cores with all four optimizations and a *TIMEOUT* threshold of 500 seconds.

Within this time limit, *SER* solved 26 of the 27 serializable benchmarks and 19 of the 20 non-serializable benchmarks (see summary in Table 1 and the full results in Table 4 of the appendix in our extended paper [9]). The *median* total runtime was 1,909 ms across all benchmarks, and 2,238.5 ms (830 ms) when solely focusing on serializable (non-serializable) benchmarks. The *average* total runtime was 32,898.38 ms across all benchmarks, and 25,530.69 ms (42,980.47 ms) when solely focusing on serializable (non-serializable) benchmarks. We also observe a clear runtime split based on serializability: among non-serializable benchmarks, counterexample generation takes much longer than validation, and dominates the overall runtime; whereas among serializable benchmarks the validation time dominates the overall runtime. This is not surprising, as validating a given counterexample only requires a polynomial-time simulation of the network system to confirm its feasibility.

Category	average (ms)		median (ms)	
	cert.	total	cert.	total
Serializable	2,273	25,531	1,178	2,239
Not serializable	42,076	42,980	773	830
All	19,079	32,898	773	1,909

Table 1: Runtime for generating certificates (**cert.**) and the overall runtime (**total**), including for validation.

Benchmark	Serializable	Features						Runtime (ms)
		If	While	?	Arith	Yield	Multi-req	
banking (g1)	✗	✓	✓		✓	✓	✓	74,539
banking (g2)	✓	✓	✓		✓	✓	✓	TIMEOUT
routing (g3)	✗	✓	✓	✓	✓	✓	✓	20,954
monitor (g4)	✗	✓	✓	✓	✓	✓	✓	7,047
monitor (g5)	✓	✓	✓	✓	✓	✓	✓	12,324
firewall (g6)	✗	✓		✓	✓	✓		8,285
firewall (g7)	✓	✓		✓	✓			252,752

Table 2: Overview of benchmarks from the *network & system protocols* category.

6 Related Work

Theoretical results. Serializability (or *atomicity*) was first introduced by Eswaran et al. [44], later motivating Herlihy and Wing’s [52,53] similar notion of *linearizability* for concurrent data structures. The *membership problem* — deciding if a specific interleaving is serializable — is NP-complete [79], a result that was later extended to linearizability [50], as well as to other consistency models [16]. The *correctness problem* — whether *all* executions satisfy this criterion — is EXPSPACE when threads are bounded [5] and undecidable otherwise [18], though decidable for bounded-barrier programs (and hence, for serializability in our setting). Bouajjani et al. [20] further show that unbounded-thread linearizability for certain ADTs reduces to VASS coverability in EXPSPACE [84]. The SER toolchain is, to our knowledge, the first to implement Bouajjani et al.’s serializability algorithm [18], adapting it to distributed transactions, extending it with a proof certificate mechanism, and scaling it with various optimizations to models of real-world programs.

Model checking and runtime verification. Runtime checks for serializability and conflict/view-serializability were proposed by Wang and Stoller [97,98]. TLA logic [65] can express various serializability forms [31], however, such approaches [54,88] remain restricted to bounded systems due to finite-state tools (TLC, Apalache [59,101]). Heuristic or enumeration-based model checkers include Line-up [24] (built upon CHESS [75]), LinTSO [25], Violat [41] and its schema precursor [40], bridge-predicate methods [26,27], and PAT-based refinement checking [69,70,89,102]. Recent work includes RELINCHE for bounded linearizability [51], CDSSpec [78] (for C/C++11), Lincheck [61], and SAT-based [15,96] approaches [23]. Symbolic testing [42] can expose violations of observational refinement [19,47]. Other checkers, e.g. SPIN/PARGLIDER [48,56,93,94], depend on explicit linearization points, which are difficult to determine [94]. Overall, existing methods are typically incomplete, bounded, or assume prior knowledge.

By contrast, our method covers unbounded threads and uniquely produces serializability certificates.

Static analysis. Static methods prove linearizability for bounded [10, 72] and unbounded systems [11, 91, 92], but usually rely on heuristics or annotated linearization points (e.g. [38]). Lian and Feng [68] propose a logic for non-fixed points. However, annotation-based analyses [4, 77, 103] may be inconclusive, as failures can stem from incorrect annotations rather than from true violations [19].

Manual proofs and additional approaches. Tasiran [90] proves serializability for **Bartok-STM**, while Colvin et al. [33] use I/O automata for list-set linearizability. Simplifications exist for specific data structures [21, 46]. Other notable linearizability results include Wing and Gong’s [99] on unbounded FIFO/priority queues, Chakraborty et al. [29] on queues, and Cerný et al.’s CoLT for linked heaps (which is complete only under bounded threads [28]). Bouajjani et al. [22] introduce a recursive priority-queue violation detector, akin to their stack/queue methods [20]. Other strategies include testing [39, 71, 81, 82, 99], theorem proving [32, 37], and the use of additional verification frameworks [21, 43, 46].

7 Discussion

7.1 Limitations

While our approach advances the state of the art in verifying unbounded serializability, several limitations remain. First, the underlying Petri net reachability problem has **Ackermann**-complete complexity [34, 66], causing our tool to time out on some complex benchmarks. Second, our current implementation relies on **SMPT** [7], which may fail to find proofs even when they exist, limiting completeness. Third, our network system model assumes a simple request/response pattern and cannot model more complex interactions, such as streaming, call-backs, or partial responses. Finally, **SER** targets finite-state programs: each request must have finite local state and the program must induce finite global state (with an unbounded number of requests). Thus, applying **SER** to real systems requires that executions generate only a finite *reachable* state space, in order for the NS construction to terminate. This setting is akin to model checkers such as **PRISM** [63] and **STORM** [36].

7.2 Future Work

Additional optimizations. To improve scalability, we are adapting *polyhedral reductions* [6, 8], a form of structural reduction [12, 13] $(N_1, m_1) \triangleright_E (N_2, m_2)$ where N_2 is a simpler Petri net and E allows reconstruction of N_1 ’s state space. This would allow verification on the reduced net, with proofs lifted back to the original one. Moreover, we believe a further avenue for optimization lies in using *approximations* to decide serializability. Our approach already leverages this idea via the underlying model checker, which employs the *state equation*

abstraction [74] to over-approximate the reachable state space. We expect that additional approximation-based techniques could yield further scalability gains. Finally, other potential optimizations involve short-circuiting steps in our algorithmic pipeline. For instance, we currently generate the reachability query $\mathcal{F}$ in three stages: (i) translating the serial NFA into a regular expression via Kleene’s theorem; (ii) translating the regular expression into a semilinear set using Parikh’s construction; and (iii) complementing the resulting semilinear set. However, there are techniques (e.g., Verma et al. [95]) that *directly* compute the Parikh image of an automaton. We did not adopt this approach because, although its construction is linear in size, it relies heavily on Boolean logic, which we found ISL (the standard integer set library) handles poorly in practice.

Proof assistants. Another natural next step is to formalize certificate checking in a proof assistant (e.g., Rocq [1]). This would entail (i) developing a verifier for the PN invariants we use; and (ii) proving theorems that connect invariant validity to serializability. Specifically, this would likely require extending existing tactics such as LIA (Linear Integer Arithmetic), which currently does not support full Presburger arithmetic, as required by our logic.

7.3 Applicability to Real-World Programs

Real-world SDN programs typically satisfy our finite-reachable-state requirement due to bounded end-host buffers and limited switch memory. Moreover, we anticipate that P4 programs [17] can be translated to SER based on the following high-level mappings: (i) packets to requests, (ii) switch registers to global variables, (iii) packet header fields to local variables, and (iv) packet forwarding to yielding. This translation motivated us to evaluate our toolchain on programs modeling *stateful firewalls* [55, 57]. Furthermore, we believe our framework is applicable beyond SDNs. Specifically, our NS model abstracts distributed state with message-passing/RPC-style concurrency, which aligns naturally with database transactions. One such example is our *snapshot-monitoring benchmark* (see Appendix A of our extended paper [9]).

7.4 Conclusion

We present the first end-to-end framework that automatically verifies serializability for unbounded concurrent systems and generates proof certificates thereof. Our approach bridges theory and practice, with the following key contributions: (1) formalizing serializability for network systems, (2) implementing the decision procedure with proof generation, (3) developing optimizations that reduce complexity by orders of magnitude, and (4) demonstrating feasibility on various benchmarks inspired by real-world systems.

Data and Software Availability

The data and software necessary to reproduce the experiments in this paper are available as part of the accompanying artifact [3].

Acknowledgements

The work of Amir was partially supported by a Rothschild Fellowship from Yad Hanadiv (The Rothschild Foundation). We thank Nate Foster, Fred B. Schneider, Lorin Hochstein, Petr Jancar, and Wolfgang Reisig for their contributions to this project.

References

1. The Rocq Prover. <https://rocq-prover.org/>, accessed: 2025-12-13
2. CockroachDB: Revision 7. <https://dbdb.io/db/cockroachdb/revisions/7> (2018), [Online; accessed 2025-10-02]
3. Supplementary Artifact (2025), <https://zenodo.org/records/17253581>
4. Abdulla, P., Jonsson, B., Trinh, C.: Automated Verification of Linearization Policies. In: Proc. 23rd Int. Symposium on Static Analysis (SAS). pp. 61–83 (2016)
5. Alur, R., McMillan, K., Peled, D.: Model-Checking of Correctness Conditions for Concurrent Objects. In: Proc. 11th ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 219–228 (1996)
6. Amat, N., Berthomieu, B., Dal Zilio, S.: On the Combination of Polyhedral Abstraction and SMT-Based Model Checking for Petri Nets. In: Proc. 42nd Int. Conf. on Applications and Theory of Petri Nets and Concurrency (PETRI NETS). pp. 164–185 (2021)
7. Amat, N., Dal Zilio, S.: SMPT: A Testbed for Reachability Methods in Generalized Petri Nets. In: Proc. 25th Int. Symposium on Formal Methods (FM). pp. 445–453 (2023)
8. Amat, N., Dal Zilio, S., Berthomieu, B.: A Polyhedral Abstraction for Petri Nets and its Application to SMT-Based Model Checking. *Fundamenta Informaticae* **187** (2022)
9. Amir, G., Barbone, M., Amat, N., Jacobs, J.: Deciding Serializability in Network Systems (2026), Technical Report. <http://arxiv.org/abs/2601.02251>
10. Amit, D., Rinetzk, N., Reps, T., Sagiv, M., Yahav, E.: Comparison Under Abstraction for Verifying Linearizability. In: Proc. 19th Int. Conf. on Computer Aided Verification (CAV). pp. 477–490 (2007)
11. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread Quantification for Concurrent Shape Analysis. In: Proc. 20th Int. Conf. on Computer Aided Verification (CAV). pp. 399–413 (2008)
12. Berthelot, G.: Transformations and Decompositions of Nets. In: Petri Nets: Central Models and their Properties. Springer (1987)
13. Berthomieu, B., Le Botlan, D., Dal Zilio, S.: Counting Petri Net Markings from Reduction Equations. *International Journal on Software Tools for Technology Transfer (STTT)* **22**, 163–181 (2020)
14. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Proc. 5th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (1999)
15. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability, vol. 185. IOS Press (2009)
16. Biswas, R., Enea, C.: On the Complexity of Checking Transactional Consistency. In: Proc. ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 1–28 (2019)

17. Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., Walker, D.: P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Computer Communication Review* **44**(3), 87–95 (2014)
18. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Verifying Concurrent Programs Against Sequential Specifications. In: *Proc. 22nd European Symposium on Programming (ESOP)*. pp. 290–309 (2013)
19. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Tractable Refinement Checking for Concurrent Objects. In: *Proc. 42nd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. pp. 651–662 (2015)
20. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: On Reducing Linearizability to State Reachability. *Information and Computation* **261**, 383–400 (2018)
21. Bouajjani, A., Emmi, M., Enea, C., Mutluergil, S.: Proving Linearizability Using Forward Simulations. In: *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*. pp. 542–563 (2017)
22. Bouajjani, A., Enea, C., Wang, C.: Checking Linearizability of Concurrent Priority Queues. In: *Proc. 28th Int. Conf. on Concurrency Theory (CONCUR)* (2017)
23. Burckhardt, S., Alur, R., Martin, M.: Checkfence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models. In: *Proc. 28th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. pp. 12–21 (2007)
24. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: A Complete and Automatic Linearizability Checker. In: *Proc. 31st ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. pp. 330–340 (2010)
25. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent Library Correctness on the TSO Memory Model. In: *Proc. 21st European Symposium on Programming (ESOP)*. pp. 87–107 (2012)
26. Burnim, J., Necula, G., Sen, K.: Specifying and Checking Semantic Atomicity for Multithreaded Programs. In: *Proc. 16th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. pp. 79–90 (2011)
27. Burnim, J., Sen, K.: Asserting and Checking Determinism for Multithreaded Programs. In: *Proc. 7th Symposium on the Foundations of Software Engineering (FSE)*. pp. 3–12 (2009)
28. Černý, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model Checking of Linearizability of Concurrent List Implementations. In: *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV)*. pp. 465–479 (2010)
29. Chakraborty, S., Henzinger, T., Sezgin, A., Vafeiadis, V.: Aspect-Oriented Linearizability Proofs. *Logical Methods in Computer Science* **11** (2015)
30. CockroachDB Contributors: GitHub Issue #14099: UPSERT Anomaly Under SNAPSHOT Isolation. <https://github.com/cockroachdb/cockroach/issues/14099> (2018), [Online; accessed 2025-07-10]
31. Cohen, A., O’Leary, J., Pnueli, A., Tuttle, M., Zuck, L.: Verifying Correctness of Transactional Memories. In: *Proc. 7th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*. pp. 37–44 (2007)
32. Colvin, R., Doherty, S., Groves, L.: Verifying Concurrent Data Structures by Simulation. *Electronic Notes in Theoretical Computer Science* **137**(2), 93–110 (2005)
33. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal Verification of a Lazy Concurrent List-Based Set Algorithm. In: *Proc. 18th Int. Conf. on Computer Aided Verification (CAV)*. pp. 475–488 (2006)

34. Czerwinski, W., Orlikowski, L.: Reachability in Vector Addition Systems is Ackermann-Complete. In: Proc. 62nd Annual Symposium on Foundations of Computer Science (FOCS). pp. 1229–1240 (2022)
35. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proc. 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 337–340 (2008)
36. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A Storm is Coming: A Modern Probabilistic Model Checker. In: Proc. 29th Int. Conf. on Computer Aided Verification (CAV). pp. 592–600 (2017)
37. Derrick, J., Schellhorn, G., Wehrheim, H.: Mechanically Verified Proof Obligations for Linearizability. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1), 1–43 (2011)
38. Drachsler-Cohen, D., Petrank, E.: LCD: Local Combining on Demand. In: Proc. 18th Int. Conf. on Principles of Distributed Systems (OPODIS). pp. 355–371 (2014)
39. Emmi, M., Enea, C.: Exposing Non-Atomic Methods of Concurrent Objects (2017), Technical Report. <http://arxiv.org/abs/1706.09305>
40. Emmi, M., Enea, C.: Sound, Complete, and Tractable Linearizability Monitoring for Concurrent Collections. In: Proc. 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL). pp. 1–27 (2018)
41. Emmi, M., Enea, C.: Violat: Generating Tests of Observational Refinement for Concurrent Objects. In: Proc. 31st Int. Conf. on Computer Aided Verification (CAV). pp. 534–546 (2019)
42. Emmi, M., Enea, C., Hamza, J.: Monitoring Refinement via Symbolic Reasoning. In: Proc. 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). pp. 260–269 (2015)
43. Enea, C., Koskinen, E.: Scenario-Based Proofs for Concurrent Objects. In: Proc. ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 1294–1323 (2024)
44. Eswaran, K., Gray, J., Lorie, R., Traiger, I.: The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM* **19**(11), 624–633 (1976)
45. Farzan, A., Madhusudan, P.: Monitoring Atomicity in Concurrent Programs. In: Proc. 20th Int. Conf. on Computer Aided Verification (CAV). pp. 52–65 (2008)
46. Feldman, Y., Enea, C., Morrison, A., Rinetzky, N., Shoham, S.: Order Out of Chaos: Proving Linearizability Using Local Views. In: Proc. Int. Symposium on Distributed Computing (DISC) (2018)
47. Filipović, I., O’Hearn, P., Rinetzky, N., Yang, H.: Abstraction for Concurrent Objects. *Theoretical Computer Science* **411**(51–52), 4379–4398 (2010)
48. Flanagan, C.: Verifying Commit-Atomicity Using Model-Checking. In: Proc. 11th Int. Symposium on Model Checking Software (SPIN). pp. 252–266 (2004)
49. Flanagan, C., Freund, S., Yi, J.: Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In: Proc. 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). pp. 293–303 (2008)
50. Gibbons, P., Korach, E.: Testing Shared Memories. *SIAM Journal on Computing* **26**(4), 1208–1244 (1997)
51. Golovin, P., Kokologiannakis, M., Vafeiadis, V.: Relinche: Automatically Checking Linearizability under Relaxed Memory Consistency. In: Proc. 52nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 2090–2117 (2025)

52. Herlihy, M., Wing, J.: Axioms for Concurrent Objects. In: Proc. 14th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 13–26 (1987)
53. Herlihy, M., Wing, J.: Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (3), 463–492 (1990)
54. Hochstein, L.: Serializability and TLA+ (Oct 2024), <https://surfingcomplexity.blog/2024/10/28/serializability-and-tla/>
55. Hogan, M., Landau-Feibish, S., Arashloo, M.T., Rexford, J., Walker, D.: Modular Switch Programming Under Resource Constraints. In: Proc. 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 193–207 (2022)
56. Holzmann, G.: The Model Checker SPIN. *IEEE Transactions on Software Engineering* **23**(5), 279–295 (1997)
57. Kim, D., Liu, Z., Zhu, Y., Kim, C., Lee, J., Sekar, V., Seshan, S.: TEA: Enabling State-Intensive Network Functions on Programmable Switches. In: Proc. Int. Conf. of the ACM Special Interest Group on Data Communication (SIGCOMM). pp. 90–106 (2020)
58. Kleene, S.C.: Representation of Events in Nerve Nets and Finite Automata, vol. 34. Princeton University Press Princeton (1956)
59. Konnov, I., Kukovec, J., Tran, T.H.: TLA+ Model Checking Made Symbolic. In: Proc. ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 1–30 (2019)
60. Kosaraju, S.: Decidability of Reachability in Vector Addition Systems. In: Proc. 14th Annual ACM Symposium on Theory of Computing (STOC). pp. 267–281 (1982)
61. Koval, N., Fedorov, A., Sokolova, M., Tsitelov, D., Alistarh, D.: Lincheck: A Practical Framework for Testing Concurrent Data Structures on JVM. In: Proc. 35th Int. Conf. on Computer Aided Verification (CAV). pp. 156–169 (2023)
62. Kreutz, D., Ramos, F., Verissimo, P.E., Rothenberg, C., Azodolmolky, S., Uhlig, S.: Software-Defined Networking: A Comprehensive Survey. *Proc. of the IEEE* **103**(1), 14–76 (2014)
63. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic Symbolic Model Checker. In: Proc. 12th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS). pp. 200–204 (2002)
64. Lambert, J.L.: A Structure to Decide Reachability in Petri Nets. *Theoretical Computer Science* **99**(1), 79–104 (1992)
65. Lamport, L.: The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (3), 872–923 (1994)
66. Leroux, J.: The Reachability Problem for Petri Nets is Not Primitive Recursive. In: Proc. 62nd Annual Symposium on Foundations of Computer Science (FOCS). pp. 1241–1252 (2022)
67. Leroux, J.: The General Vector Addition System Reachability Problem by Presburger Inductive Invariants. *Logical Methods in Computer Science (LMCS)* (2010)
68. Liang, H., Feng, X.: Modular Verification of Linearizability with Non-Fixed Linearization Points. In: Proc. 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). pp. 459–470 (2013)
69. Liu, Y., Chen, W., Liu, Y., Sun, J.: Model Checking Linearizability via Refinement. In: Proc. 16th Int. Symposium on Formal Methods (FM). pp. 321–337 (2009)

70. Liu, Y., Chen, W., Liu, Y., Sun, J., Zhang, S.J., Dong, J.S.: Verifying Linearizability via Optimized Refinement Checking. *IEEE Transactions on Software Engineering* **39**(7), 1018–1039 (2012)
71. Lowe, G.: Testing for Linearizability. *Concurrency and Computation: Practice and Experience* **29**(4), e3928 (2017)
72. Manevich, R., Lev-Ami, T., Sagiv, M., Ramalingam, G., Berdine, J.: Heap Decomposition for Concurrent Shape Analysis. In: *Proc. 15th Int. Symposium on Static Analysis (SAS)*. pp. 363–377 (2008)
73. Mayr, E.: An Algorithm for the General Petri Net Reachability Problem. In: *Proc. 13th Annual ACM Symposium on Theory of Computing (STOC)*. pp. 238–246 (1981)
74. Murata, T.: State Equation, Controllability, and Maximal Matchings of Petri Nets. *Transactions on Automatic Control* **22**, 412–416 (1977)
75. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and Reproducing Heisenbugs in Concurrent Programs. In: *Proc. 8th USENIX Symposium on Operating Systems Design and Implementations (OSDI)* (2008)
76. Namjoshi, K., Gheissi, S., Sabnani, K.: Algorithms for In-Place, Consistent Network Update. In: *Proc. Int. Conf. of the ACM Special Interest Group on Data Communication (SIGCOMM)*. pp. 244–257 (2024)
77. O’Hearn, P., Rinetzk, N., Vechev, M., Yahav, E., Yorsh, G.: Verifying Linearizability with Hindsight. In: *Proc. 29th Symposium on Principles of Distributed Computing (PODC)*. pp. 85–94 (2010)
78. Ou, P., Demsky, B.: Checking Concurrent Data Structures Under the C/C++ 11 Memory Model. In: *Proc. 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. pp. 45–59 (2017)
79. Papadimitriou, C.: The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM)* **26**(4), 631–653 (1979)
80. Parikh, R.J.: On Context-Free Languages. *Journal of the ACM (JACM)* **13**(4), 570–581 (1966)
81. Pradel, M., Gross, T.: Fully Automatic and Precise Detection of Thread Safety Violations. In: *Proc. 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. pp. 521–530 (2012)
82. Pradel, M., Gross, T.: Automatic Testing of Sequential and Concurrent Substitutability. In: *Proc. 35th Int. Conf. on Software Engineering (ICSE)*. pp. 282–291 (2013)
83. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchen die Addition als einzige Operation hervortritt. In: *Comptes-rendus du 1er congrès des mathématiciens des pays slaves* (1929)
84. Rackoff, C.: The Covering and Boundedness Problems for Vector Addition Systems. *Theoretical Computer Science* **6**(2), 223–231 (1978)
85. Rakow, A.: Safety Slicing Petri Nets. In: *Proc. 33rd Int. Conf. on Applications and Theory of Petri Nets and Concurrency (PETRI NETS)*. pp. 268–287 (2012)
86. Sinha, A., Malik, S., Wang, C., Gupta, A.: Predicting Serializability Violations: SMT-Based Search vs. DPOR-Based Search. In: *Proc. 7th Haifa Verification Conference (HVC)*. pp. 95–114 (2011)
87. Sinha, A., Malik, S., Wang, C., Gupta, A.: Predictive Analysis for Detecting Serializability Violations Through Trace Segmentation. In: *Proc. 9th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign (MEMPCODE)*. pp. 99–108 (2011)

88. Soethout, T., van der Storm, T., Vinju, J.J.: Automated Validation of State-Based Client-Centric Isolation with TLA+. In: Proc. 18th Int. Conf. Software Engineering and Formal Methods (SEFM). pp. 43–57 (2020)
89. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification Under Fairness. In: Proc. 21st Int. Conf. on Computer Aided Verification (CAV). pp. 709–714 (2009)
90. Tasiran, S.: A Compositional Method for Verifying Software Transactional Memory Implementations. Microsoft Research, Technical Report MSR-TR-2008–56 (2008)
91. Vafeiadis, V.: Shape-Value Abstraction for Verifying Linearizability. In: Proc. 9th Int. Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI). pp. 335–348 (2009)
92. Vafeiadis, V.: Automatically Proving Linearizability. In: Proc. 22nd Int. Conf. on Computer Aided Verification (CAV). pp. 450–464 (2010)
93. Vechev, M., Yahav, E.: Deriving Linearizable Fine-Grained Concurrent Objects. In: Proc. 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). pp. 125–135 (2008)
94. Vechev, M., Yahav, E., Yorsh, G.: Experience with Model Checking Linearizability. In: Proc. 16th Int. Symposium on Model Checking Software (SPIN). pp. 261–278 (2009)
95. Verma, K.N., Seidl, H., Schwentick, T.: On the Complexity of Equational Horn Clauses. In: Proc. 20th Int. Conf. on Automated Deduction (CADE). pp. 337–352 (2005)
96. Vizel, Y., Weissenbacher, G., Malik, S.: Boolean Satisfiability Solvers and their Applications in Model Checking. Proc. of the IEEE **103**(11), 2021–2035 (2015)
97. Wang, L., Stoller, S.: Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs. In: Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). pp. 137–146 (2006)
98. Wang, L., Stoller, S.: Runtime Analysis of Atomicity for Multithreaded Programs. IEEE Transactions on Software Engineering **32**(2), 93–110 (2006)
99. Wing, J., Gong, C.: Testing and Verifying Concurrent Objects. Journal of Parallel and Distributed Computing **17**(1-2), 164–182 (1993)
100. Xia, W., Wen, Y., Foh, C.H., Niyato, D., Xie, H.: A Survey on Software-Defined Networking. IEEE Communications Surveys & Tutorials **17**(1), 27–51 (2015)
101. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA+ Specifications. In: Proc. Int. Conf. Correct Hardware Design and Verification Methods (CHARME). pp. 54–66 (1999)
102. Zhang, S.J.: Scalable Automatic Linearizability Checking. In: Proc. 33rd Int. Conf. on Software Engineering (ICSE). pp. 1185–1187 (2011)
103. Zhu, H., Petri, G., Jagannathan, S.: Poling: SMT Aided Linearizability Proofs. In: Proc. 27th Int. Conf. on Computer Aided Verification (CAV). pp. 3–19 (2015)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



VERISTRUCT: AI-assisted Automated Verification of Data-Structure Modules in Verus

Chuyue Sun¹, Yican Sun²^{*}, Daneshvar Amrollahi¹, Ethan Zhang¹,
Shuvendu Lahiri³, Shan Lu³, David Dill¹, and Clark Barrett¹

¹ Stanford University

² School of Computer Science, Peking University, Beijing, China

³ Microsoft Research

Abstract. We introduce VERISTRUCT, a novel framework that extends AI-assisted automated verification from single functions to more complex data structure modules in Verus. VERISTRUCT employs a planner module to orchestrate the systematic generation of abstractions, type invariants, specifications, and proof code. To address the challenge that LLMs often misunderstand Verus’ annotation syntax and verification-specific semantics, VERISTRUCT embeds syntax guidance within prompts and includes a repair stage to automatically correct annotation errors. In an evaluation on eleven Rust data structure modules, VERISTRUCT succeeds on ten of the eleven, successfully verifying 128 out of 129 functions (99.2%) in total. These results represent an important step toward the goal of automatic AI-assisted formal verification.

1 Introduction

The power of generative AI introduces new risks to our critical digital infrastructure. AI systems have a remarkable ability to analyze code, making it much easier for a malicious actor to find and exploit vulnerabilities [3]. At the same time, more and more code is being written by or with the assistance of AI systems. While it is true that such AI systems can greatly increase the productivity of programmers, they can also introduce additional correctness errors and security vulnerabilities [25, 36]. Thus, productivity gains come at the risk of churning out mountains of buggy, insecure code.

Program verification can mitigate these risks by mathematically proving that code—whether human-written or AI-generated—is free of bugs and vulnerabilities. However, program verification relies on the addition of sophisticated *logical annotations* to code, including preconditions, postconditions, invariants, and auxiliary proof blocks. These annotations, which we also refer to as *specifications*, are used to generate verification conditions, which are then proved automatically by automated reasoning tools like satisfiability modulo theories (SMT) solvers [4]. The application of program verification in practice has been severely limited because of the enormous effort and expertise required to create such annotations.

^{*} Work was accomplished while the author was visiting Stanford University.

Fortunately, the power of generative AI is also shifting the calculus on the practicality of program verification. Recently, there have been encouraging results on using large language models (LLMs) to *automatically* specify and verify software [20, 29, 30, 37, 39]. Our work aims to build on these efforts with the vision of one day making it possible to have AI-assisted program verification at scale and verified libraries of reusable code [34].

So far, AI-assisted verification efforts have focused on simple textbook algorithms consisting of a single function. In this paper, we extend this by developing a novel workflow for the verification of data structure modules. The ability to verify data structures is a crucial advance since: (1) data structures are ubiquitous in everyday programming, forming the foundation of many software systems; and (2) verifying library data structures lowers the burden for verifying more complex code, as client code can then rely on the correctness of these already-proven components.

We implement our workflow in VERISTRUCT, an AI-assisted automated verification framework for Verus [12, 13]. Verus is a state-of-the-art program verification extension for Rust. We target Verus because it has been successfully applied to the construction of verified systems [12], and its design allows developers to write verification annotations using a syntax closely aligned with Rust. This design not only lowers the barrier for Rust engineers to adopt formal verification but also positions Verus to have an increasing impact as Rust continues to gain traction in mainstream software development.

The input to VERISTRUCT consists of the Rust source code for the module to be verified and a unit test suite that illustrates the intended usage and expected behavior of the data structure. The test suite serves a dual purpose: it helps ensure that the generated formal specifications align with the developer’s intent, and it helps rule out trivial or vacuous specifications [11]. VERISTRUCT automatically augments the provided Rust implementation with annotations needed for program verification. This augmented code, together with the provided unit test suite, are then sent to the Verus verifier, which attempts to formally verify the assertions in the test suite. If the verification check succeeds, the formally verified code is returned to the user. Otherwise, VERISTRUCT attempts to extend or repair the annotations based on the feedback from the verifier.

We highlight two challenges for LLM-assisted verification of data-structure modules. The first challenge is that verifying data structures is inherently more complex than verifying a single function. Specifically, data-structure verification often requires two additional elements: (i) a suitable mathematical abstraction that allows the verifier to reason logically about the data structure [22, 24, 27]; and (ii) so-called *type invariants* that must be preserved by operations on the data structure [19, 29]. Moreover, data-structure modules usually expose multiple methods that must be verified jointly under a shared type invariant. This stands in contrast to other recent approaches, which focus on generating a single specification artifact (e.g., proof blocks for an individual function or automated synthesis of class invariants) [9, 20, 26, 29, 37–39]. In short, our verification task extends beyond the scope of these existing techniques.

The second challenge lies in the LLMs’ limited understanding of Verus’ specialized syntax and verification-specific semantics. For instance, annotations in Verus are only allowed to invoke “specification functions,” which cannot modify global state. However, the LLM sometimes suggests invoking a regular Rust function. Such deficiencies primarily stem from the scarcity of Verus code in existing training data.

To address the first challenge, we build on previous work [39] by adding two new modules: (1) a *View* module responsible for producing a mathematical abstraction of the data structure; and (2) a *Type Invariant* module, responsible for producing type invariants. For each component, we design dedicated system and user prompts that clearly articulate the task. These prompts include: (1) the objective of the component; (2) relevant background information to help the LLM better understand the task; (3) step-by-step instructions and the required output format; and (4) in-context learning examples [6]. Additionally, because not all verification components are necessary for every task, we introduce a *planner* module, which analyzes the verification task as an initial step and decides which components need to be generated.

To address the second challenge, we pursued two approaches. First, we developed a suite of automated repair modules to identify and correct errors in the generated annotations. Compared with the previous work [39] which focused solely on repairing proof blocks, we further introduce dedicated repair modules targeting views, type invariants, and specifications. Specifically, to facilitate the joint verification of multiple methods and tests, VERISTRUCT includes a repair module that refines specifications whenever the verifier reports a failed test verification. Second, we embedded structured syntax guidelines into the prompt to mitigate syntax errors in generated annotations. We sourced these guidelines from the Verus tutorial [32] and from standard Verus libraries [33].

We evaluate VERISTRUCT on eleven data-structure benchmarks drawn from Verus examples and third-party code. Our framework automatically produces verified implementations for ten of these benchmarks while substantially reducing the amount of handwritten annotation required from developers. When writing unit tests, we observe that achieving high coverage is both challenging and labor-intensive. To alleviate this burden, we apply LLM-assisted techniques to automatically generate unit test cases.

To summarize, this paper makes the following key contributions:

1. we introduce a new LLM-assisted workflow for generating program verification annotations for data-structure modules;
2. we provide an implementation of our workflow in the VERISTRUCT tool; and
3. we evaluate VERISTRUCT on eleven data structure benchmarks, demonstrating its effectiveness.

The prompt and source code of VERISTRUCT is available on GitHub [1].

```

1  verus! { // Indicates the Verus environment
2
3  pub struct RingBuffer<T: Copy> {
4      ring: Vec<T>,
5      head: usize,
6      tail: usize,
7  }
8  // The View trait, which tells the Verus verifier
9  // how to logically represent RingBuffer
10 impl<T: Copy> View for RingBuffer<T> {
11     type V = (Seq<T>, usize);
12     closed spec fn view(&self) -> Self::V {
13         let cap = self.ring.len();
14         let content =
15             if self.tail >= self.head {
16                 (self.ring).0.subrange(
17                     self.head as int, self.tail as int
18                 )
19             } else {
20                 (self.ring).0.subrange(
21                     self.head as int, cap as int
22                 ).add((self.ring).0.subrange(
23                     0, self.tail as int
24                 ))
25             };
26         (content, cap)
27     }
28 }

28 impl RingBuffer<T> {
29     // The type invariant, specifying the properties that
30     // RingBuffer object must satisfy
31     #[verifier::type_invariant]
32     closed spec fn inv(&self) -> bool {
33         &&& self.head < self.ring.len()
34         &&& self.tail < self.ring.len()
35         &&& self.ring.len() > 0
36     }
37     pub fn new(ring: Vec<T>) -> (ret: RingBuffer<T>)
38     requires
39         ring.len() >= 1
40     ensures
41         ret.0.len() == 0,
42         ret.1 == ring.len() as nat
43     {
44         // Create an empty ring buffer
45         RingBuffer { head: 0, tail: 0, ring }
46     }
47     pub fn is_full(&self) -> (ret: bool)
48     ensures
49         ret == (self.0.len() == (self.1 - 1) as nat)
50     {
51         proof {
52             use_type_invariant(&self);
53         }
54         self.head == ((self.tail + 1) % self.ring.len())
55     }
56     pub fn enqueue(&mut self, val: T) -> (succ: bool) { ... }
57     pub fn dequeue(&self) { ... }
58 }
59 fn test_ring_buffer() { /* Unit tests (omitted). */ }
60 fn main() { test_ring_buffer(); }
61 } // End of verus!

```

Fig. 1. Verified Ring Buffer, Lines Highlighted in Green are Annotations

2 Preliminaries: Verifying a Data-Structure in Verus

In this section, we provide background on deductive verification and Verus. We start by introducing an example that will be used to illustrate key concepts throughout the paper.

Example 1 (Verified Ring Buffer). A ring buffer is a fixed-size data structure that treats memory as a circular array. Ring buffers are widely used in streaming, networking, and real-time systems [10, 16]. Fig. 1 shows the Rust implementation for a ring buffer, including the Verus annotations necessary for verifying key properties of the implementation. $\square$

Generally speaking, deductive program verification augments programs with *logical annotations*. These annotations connect directly to correctness: the verifier generates proof obligations from the code and annotations; if all obligations are proved, then the implementation is correct with respect to the annotations. Otherwise, the verifier produces an error message. Below, we describe the different categories of annotations available in the Verus program verification framework.

Specification Functions. In Verus, functions are classified into two categories: *specification functions* and *executable functions*. Specification functions, marked with the `spec` keyword (e.g., `view` and `inv` in Fig. 1), are pure and cannot modify any global or mutable state. They are simply logical definitions of pure functions without side effects and are called exclusively from specifications, invariants, and proofs. In contrast, executable functions (e.g., `new`, `is_full`, `enqueue`, and `dequeue`) define concrete program behavior and may modify state. When writing

annotations, only specification functions can be called, since annotations must remain side-effect-free and semantically independent of execution.

Pre- and Postconditions. Annotations include preconditions and postconditions for each function. The implicit program requirement is that: if a function is invoked with a program state satisfying the precondition, then it will return (or terminate) in a state satisfying the postcondition. In Verus, one uses the keywords **requires** (Lines 38–39) and **ensures** (Lines 40–41, 48–49) for specifying preconditions and postconditions, respectively. The verifier checks that every call site meets the callee’s preconditions and that every function body establishes its postconditions from the assumed preconditions.

Proof Blocks. Implementation code may include *proof blocks* embedded in the code. These blocks provide hints to the verifier to help it discharge difficult proof obligations (Lines 51–53 in Fig. 1).

Verus includes two additional constructs that are helpful when verifying data structure modules: views and type invariants.

View Traits. The *view trait* in Verus establishes a bridge between the concrete implementation of a data structure and its abstract mathematical representation used in specifications. It requires implementing a **view()** function, which defines how an instance of a data structure maps to a logical object that the verifier can reason about. While one could theoretically encode a data structure’s state as a Cartesian product of all its fields, defining a custom view offers better abstraction, clarity, and proof efficiency, allowing specifications and proofs to focus on the logical behavior rather than the implementation details.

Once the view is implemented, specifications can refer to the logical abstraction using **a.view()** or its shorthand **a@**. Verus provides built-in specification types such as **nat**, **Seq<T>**, **Set<T>**, and **Map<K,V>** to represent natural numbers, sequences, sets, and maps, respectively. These types serve as good building blocks for implementing views.

Example 2. In Fig. 1, the **View** for **RingBuffer** (Lines 8–28) abstracts the buffer as a pair (**Seq<T>**, **usize**). The first element in the pair is a mathematical sequence (**Seq<T>**) that reflects the elements currently stored in the buffer in order, starting from the element at **head** and continuing up to (but not including) the element at **tail**. The second element in the pair is the capacity of the ring buffer. To implement the view function, we slice and concatenate segments of the underlying ring according to the positions of **head** and **tail**, while hiding **head** and **tail** themselves.

This logical view abstracts the circular implementation, enabling specifications for buffer operations—like **enqueue** and **dequeue**—to be written as simple sequence transformations (e.g., appending or removing elements), without dealing with low-level index arithmetic or memory layout details.

Type Invariants. A *type invariant* is a logical formula that all instances of a data structure must satisfy. In Verus, one declares a type invariant using the

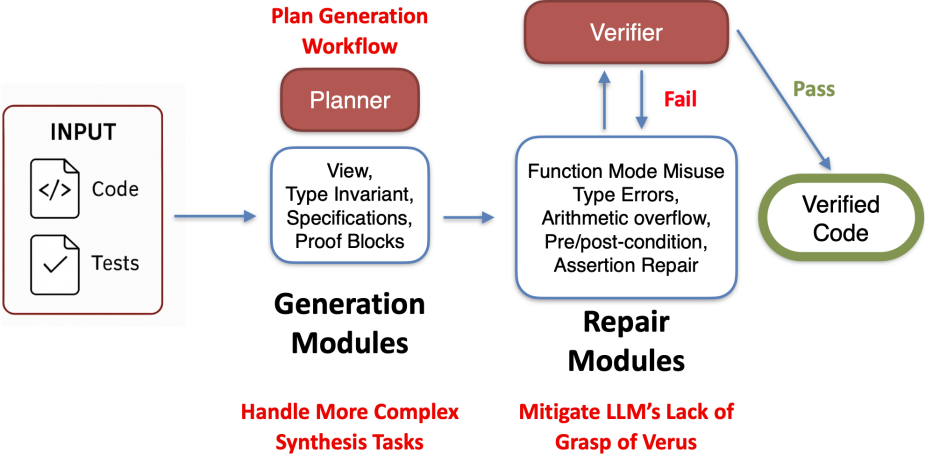


Fig. 2. Workflow of VERISTRUCT, where rounded rectangles denote modules

`#[verifier::type_invariant]` attribute. We can call `use_type_invariant` (Line 51) within a proof block to make invariant facts available in the current proof context.

Example 3. In Fig. 1 (Lines 29–36), the ring buffer invariant states that `head` and `tail` are valid indices and that the ring has positive capacity.

3 The VERISTRUCT Workflow

In this section, we give an overview of the workflow used in VERISTRUCT using the `RingBuffer` (Example 1) as a running example.

The input to VERISTRUCT consists of two parts: (1) a user-provided Rust implementation of the code to be verified (e.g., the code in Fig. 1 but *without* any annotations—the green-highlighted lines would be absent), and (2) a unit-test suite that conveys typical usage patterns and rules out trivial specifications. The output is a fully annotated version of the code that passes the Verus verifier (e.g., the full `RingBuffer` code as shown in Fig. 1). Care is taken to ensure that both the code and the tests remain unchanged throughout the workflow.

The overall workflow is shown in Fig. 2. VERISTRUCT builds on the general approach of the AutoVerus system [39], in that it consists of a network of modules that collaborate to complete program verification tasks. However, as noted in the challenge discussion of Sect. 1, verifying data structures requires techniques and components not found in AutoVerus or any prior work [20, 26, 29, 37, 39].

VERISTRUCT applies a two-stage pipeline as shown in Algorithm 1 to synthesize annotations and perform verification.

1. Stage 1: Generate the initial draft of annotations through the function **GenAnnos**. The detailed process is described in Sect. 4.

Algorithm 1: The Outermost Algorithm of VERISTRUCT

```

1 Input. code: the data structure code to be verified; test: unit test suite
2 Output. verified version of code
3 Function VerifyModule(code, test)
4   code  $\leftarrow$  GenAnnos(code, test) // Stage 1: Generate the Annotations
5   return RepairAnnos(code, test) // Stage 2: Repair Annotations

```

Algorithm 2: The Generation Stage

```

1 Parameter. n: Number of samples generated in each module invocation.
2 Input. code: the data structure code to be verified; test: unit test suite
3 Output. code with annotations generated
4 Function GenAnnos(code, test)
5   plan  $\leftarrow$  ExecPlanner(code)
6   foreach  $M_i \in \text{plan}$  do
7     code  $\leftarrow$  ExecWithSampling( $M_i$ , code, test, n)
8   return code

```

2. Stage 2: Invoke **RepairAnnos** to iteratively detect and repair incorrect annotations. The details are presented in Sect. 5.

Due to space limitations, we only present sketches of our prompts, but the full details of our prompts are publicly available [1].

4 Stage 1: Generating the Annotations

The generation stage creates annotations. As mentioned above, we may need to generate different kinds of annotations, so VERISTRUCT uses four dedicated modules, one for each type of annotation: (M_1) a view module, (M_2) a type invariant module, (M_3) a specifications module (for specification functions, preconditions, and postconditions), and (M_4) a proof blocks module, which generates both proof blocks and invariants.

By examining the dependencies among these kinds of annotations, we observe that it should typically be sufficient to invoke the modules in order: $\langle M_1, M_2, M_3, M_4 \rangle$. At the same time, it is not always necessary to execute every module. For example, some data structures do not require non-trivial type invariants; in such cases, generating an invariant may incur: (1) more instability in the language model (and thus affect the robustness of our framework); (2) unnecessary computational cost, and (3) more runtime overhead. To mitigate these issues, we employ a *planner* module that determines which generation modules should be executed.

Planner. The generation process begins with the planner agent (Line 5). The function **ExecPlanner** accepts the target code as input, invokes the planner agent, and outputs the modules to execute. The planner agent instructs the LLM to select only the necessary generation modules to be executed. Its prompt

comprises four blocks: (i) a concise description of the role of the planner; (ii) a catalog of the available generation modules: there are four in our current framework; (iii) background knowledge on Verus annotation generation; and (iv) a machine-parseable output format for the execution sequence.

The prompt is designed with several guiding principles:

1. Invoke M_1 if the data structure can be represented by a sequence, set, or map.
2. Invoke M_2 when non-trivial relationships exist among fields (e.g., range constraints or arithmetic relations).
3. If M_2 is invoked, also execute M_4 to add type invariants into the proof context.

For the ring buffer, the planner emits the full execution sequence, namely $\langle M_1, M_2, M_3, M_4 \rangle$.

Module Invocation. Following the generated plan, the framework sequentially invokes each module M_i (Lines 6–10). The function **ExecWithSampling** takes as input: (1) a module to be executed; (2) the code to be verified; and (3) the unit test suite. It executes the module, and returns the annotated code. To improve robustness, it asks for n samples during each module invocation [28]. It then selects the result yielding the maximum number of successfully verified functions. We describe the view generation module in detail below, and then describe some differentiating aspects of the other modules, which are otherwise similar.

View Generation (and Refinement) Module. To generate a view, our prompt (Fig. 3) is structured as follows:

1. *Objective.* This part articulates the task of generating a **View** implementation.
2. *Verus Guidelines.* This part provides conceptual foundations for generating a valid **View** implementation. As discussed in Sect. 1, LLMs often struggle with Verus’ syntax and verification-specific semantics. To mitigate this, our framework incorporates guidelines on Verus syntax and semantics imported from the Verus standard library documentation [33] and the official Verus tutorial [32]. For each different module in VERISTRUCT, a specific set of guidelines is added to the prompt. For **View** generation, for example, we include:
 - An overview of the **View** trait (as introduced in Sect. 2).
 - Guidelines on how to manipulate Verus’ logical types, which serve as foundational building blocks for providing a **View** implementation.
 - A chapter on Verus’ specialized logical syntax.
3. *Step-by-Step Instructions.* This part provides step-by-step procedural instructions on how to systematically generate a view.
4. *Examples* [6]. This part provides examples of **View** implementations for other verified data structures (e.g., doubly-linked lists). These examples are kept the same across (and are disjoint from) all benchmarks.
5. *Code with Tests.* Finally, the prompt adds the code to be verified code along with the unit test suite.

```

[Objective]
You are an expert in Verus (verifier for rust). Your task is to
generate a View trait for the given data-structure module...
[Verus Guidelines]
Background on View and relevant Verus features...
[Step-by-Step Instructions]
1. Infer what should be contained in the View trait.
2. Generate the view based on the inferred information.
[Examples]
The View trait for other data structures, e.g., doubly-linked list.
[Code and Tests]
Code to be generated a View trait...

```

Fig. 3. The Prompt for View Generation

For the RingBuffer example, LLMs sometimes produce a trivial Cartesian-product view, which includes all of the data fields in RingBuffer (see Fig. 4), i.e., the view function simply maps the buffer content `self.ring` to its logical abstraction `self.ring.view()` and converts `head` and `tail` into logically defined natural numbers. This view function, while syntactically valid, fails to abstract away the circular structure of the ring buffer. As a result, the subsequent specification generation becomes significantly more complex: we have to explicitly reason about the arithmetic relationship between `head` and `tail` when specifying the behavior of operations such as `enqueue` and `dequeue`.

```

1 impl View for RingBuffer<T> {
2     type V = (Seq<T>, nat, nat);
3     closed spec fn view(&self) -> Self::V {
4         (self.ring.view(), head as nat, tail as nat)
5     }
6 }

```

Fig. 4. Result of View Generation for RingBuffer

To address this issue, we introduce an additional *view refinement* step that prompts the LLM to reconsider the generated **View** and reconstruct it with fewer, more abstract components. This module encourages the model to focus on the conceptual essence of the data structure rather than on reproducing its concrete implementation. The refinement process substantially simplifies subsequent specification and proof generation tasks.

The *view refinement* prompt follows the same structure as Fig. 3, but emphasizes abstraction, minimality, and logical coherence over completeness. After applying this refinement, the model typically is able to produce an improved **View** implementation, like the one shown in Fig. 1.

Other Modules. The prompts for the other modules share a similar structure to that shown in Fig. 3. Each module prompt begins with a task description,

followed by module-specific Verus guidelines. These guidelines are specialized to the individual modules.

The *Type Invariant Module* is focused on generating type invariants, which were introduced in Sect. 2. In this module, the LLM is instructed to focus on common patterns such as range constraints, capacity checks, and arithmetic relationships between fields.

The *Specification Module* is focused on generating specification functions, preconditions and postconditions. We add mutability guidelines to ensure compatibility with Rust’s ownership and mutability system in **requires** and **ensures** clauses. For example, the LLM is prompted to use **old(·)** in **requires** to reference a variable’s value prior to function invocation.

The *Proof Block Module* handles proof blocks and loop invariants. The LLM is directed to construct the appropriate proof blocks and loop invariants, while explicitly incorporating type invariants into the proof context or applying relevant lemmas when necessary.

5 Stage 2: Repairing the Annotations

Unfortunately, a single generation run rarely yields fully verified code. LLMs often struggle with Verus’ syntax and rules. For example, the following is an *incorrect* specification that was generated for the **enqueue** method of the **RingBuffer** example:

```

1 pub fn enqueue(&mut self, val: T) -> (succ: bool)
2 ensures
3     succ == !old(self).is_full(),
4     ... // omitted
5 { /* omitted */ }
```

The intent is reasonable: **enqueue** succeeds if and only if the original buffer is not full. However, **is_full** is an *executable* function (see Line 46 of Fig. 1) and therefore cannot be called from a specification context. The verifier reports:

Error: cannot call the executable function **is_full** in annotation.

To handle such issues, VERISTRUCT executes an iterative repair loop. In each iteration, it: (i) selects an error reported by the verifier; (ii) applies a corresponding predefined repair module; and (iii) reruns verification. The process continues until either all errors are eliminated or a preset iteration budget is exhausted. VERISTRUCT provides a suite of repair modules, each targeting a specific class of errors. Error messages are routed to the appropriate module via lightweight pattern matching over the verifier’s error message.

For the error reported above, VERISTRUCT invokes a dedicated mode-repair module, which provides instructions for fixing this mode misuse (Figure 5). After applying this module, a new specification is generated. For example, one possible solution (one we actually observed) is to create a specification version of the **is_full** function and use it to replace all occurrences of **is_full**. With this change, the error no longer occurs, and the workflow can continue.

```
1 spec fn is_full_spec(&self) -> bool { self@0.len() == self@().1 - 1 }
```

[Objective]
Fix the error for the following code. The error indicates that an executable function is called from annotations or vice versa...

[Relevant Background]
Background on mode and relevant Verus Features

[Code with Error, and Tests] ...

[Instructions] Make one of these changes:
1. Adjust the function to be compatible with the calling context
...

Fig. 5. The Prompt for Repairing the Misuse of Specification and Executable Functions

Algorithm 3: The Repair Stage

```
1 Parameter. n: Number of samples generated in each module invocation.
2 Parameter. m: Maximum number of iterations in the repair loop.
3 Given. rep_modules: A pre-defined set of repair modules.
4 Input. code: the code after generating annotations; test: unit test suite
5 Output. code with annotations repaired
6 Function RepairAnnos(code, test)
7   for r ← 1 to m do
8     err ← TryVerify(code, test)
9     if err is None then
10      return Success, code
11     cur_module ← default_module
12     foreach (pattern, module) ∈ rep_modules do
13       if err matches pattern then
14         cur_module ← module
15         break
16     code ← ExecWithSampling(cur_module, code, test, n)
17   return Failure, code
```

Algorithm 3 shows the repair algorithm in detail. VERISTRUCT provides a predefined set of repair modules, denoted as `rep_modules`. Each module is associated with a regular expression `pattern` specifying the class of error messages that it addresses. In the current implementation, VERISTRUCT includes repair modules for: (1) misuse between specification and executable functions (as demonstrated above), (2) inconsistencies with Rust’s mutability type system, (3) violations of preconditions and postconditions, (4) arithmetic overflow or underflow, (5) mismatches between logical and Rust types, (6) test assertion failures, and (7) a fallback module, `default_module`, which prompts the LLM to attempt a repair based directly on the provided error message.

Note that the repair module for test assertion failures requires the LLM to do interprocedural analysis, something largely absent from previous work. The repair module inspects the failing assertion, identifies the method invoked immediately before it, and then instructs the LLM to strengthen that method’s postcondition so as to ensure the assertion holds.

The repair process proceeds iteratively, with a maximum number of iterations defined by the parameter `m`. In each iteration, the system first attempts to verify the current implementation via **TryVerify** (Line 8), which takes the code and test suite as input and returns the highest priority error message `err` produced by Verus. If no error is reported, verification succeeds and the code is returned (Lines 9–10). Otherwise, `err` is matched against all module patterns; if a match is found, the corresponding repair module is executed (Lines 13–15). If no match is found, the `default_module` is invoked. Each module execution includes `n` sampling attempts to improve robustness (Line 16). If the implementation remains unverified after `m` iterations, the system reports failure and returns the final unverified result (Line 17).

6 Evaluation

In this section, we describe an evaluation of VERISTRUCT on a set of Rust benchmarks.

Benchmarks. We evaluate VERISTRUCT on a benchmark set comprising eleven Rust data structure modules drawn from the Verus GitHub repository [13] as well as other open-source repositories. A list of the benchmarks is shown in Table 1. Although the original sources are publicly available, our benchmarks have been substantially modified with more complete specifications, additional methods, and unit tests. We verified that the versions used in our evaluation do not appear verbatim in the o1-2024-12-17 training snapshot; for example, no form of our fully verified RINGBUFFER appears in the dataset. Furthermore, even if older versions existed in pretraining data, our baseline results demonstrate that LLMs still struggle to synthesize correct Verus specifications without structured guidance—underscoring the necessity of VERISTRUCT’s workflow. The benchmarks cover a set of commonly-used data structures, including a ring buffer, a vector implementation with set abstractions and common algorithms, a polymorphic option type, tree-based modules for standalone nodes and a treemap, a bitmap with 64-bit blocks, and concurrent data structures such as a read–write lock and an invariant framework for shared state. Each benchmark contains 5–21 functions to verify, where the number of functions refers to the total number of member methods in the data structure module and the corresponding test functions. In total, our benchmark set includes 129 functions to be verified across all modules.

Baseline. To demonstrate the effectiveness of VERISTRUCT, we compare it against a baseline that iteratively invokes an LLM to generate annotations, without employing the systematic generation-and-repair workflow described in Sect. 3. In each iteration, the baseline calls a simple `BaselineModule` that invokes

Table 1. Benchmark set consisting of eleven data structures.

Benchmark	Description	#Functions
ATOMICS	An atomic counter in concurrent programming.	11
BITMAP	Bitmap implemented over 64-bit words.	14
TREEMAP	BST-based map with ordered key/value bindings.	21
INVARIANTS	Concurrent routines showing reusable invariants.	7
NODE	Implementing the node class of the binary search tree.	12
OPTION	Polymorphic option wrapper with safe accessor methods.	15
RINGBUFFER	Implementing a ring buffer.	13
RWLOCKVSTD	Read/write lock implementation.	5
SETFROMVEC	Set abstraction constructed from backing vectors.	10
TRANSFER	Account transfer routines that enforce balanced updates.	5
VECTORS	Implementing a vector with basic algorithms.	16

Table 2. The Comparison between VERISTRUCT and Baseline.

	#Solved	#Functions
VERISTRUCT	10 (↑ 150.0%)	128 (↑ 146.2%)
Claude Code	8	102
BASELINE	4	52

the LLM once to generate all annotations simultaneously. If the Verus verifier reports an error, we record the failing code and the associated error message, which are then injected into the prompt for subsequent iterations. The baseline prompt includes: (1) the code to be verified, (2) the unit test suite, and (3) the previously failing code along with any corresponding error message.

We also compare against Claude Code [2], a state-of-the-art coding agent that can autonomously invoke external tools such as the Verus verifier. We configure Claude Code (using Claude Sonnet 4.5) to iteratively generate annotations and respond to Verus error messages in an agentic loop, analogous to our baseline but with the added capability of tool invocation and autonomous iteration.

Procedure. We run VERISTRUCT and the baseline on every benchmark individually. For each benchmark we first execute VERISTRUCT to completion and then record how many LLM invocations it required. We observe that the maximum number of LLM invocations for VERISTRUCT is 13, so we run the baseline on each benchmark with 13 LLM invocations. All experiments are conducted on a server running Ubuntu 22.04.5 LTS with a 24-core Intel(R) Core(TM) i9-12900K CPU and 64 GB RAM. We use OpenAI o1 [23] as the back-end model for both VERISTRUCT and the baseline. We instantiate VERISTRUCT with $n=3$ and $m=5$; that is, the number of samples generated per module is 3, and the maximum number of repair rounds is set to 5.

Results. We summarize the results in Tables 2 and 3. VERISTRUCT successfully solves 10 out of 11 benchmarks. For the only unsolved benchmark (NODE), VERISTRUCT still verifies 11 out of 12 functions. In contrast, the baseline solves

only 4 out of 11 benchmarks. In terms of the total number of functions verified, VERISTRUCT verifies 128 out of 129 (99.2%) functions, whereas the baseline verifies only 52. Claude Code, despite using a more recent model (Claude Sonnet 4.5) and having the ability to autonomously invoke Verus, verifies 102 functions across 8 benchmarks—substantially better than the single-prompt baseline but still short of VERISTRUCT’s performance. Notably, Claude Code consumes approximately 24k tokens per benchmark on average, while VERISTRUCT uses only 22k tokens, demonstrating that our structured workflow achieves better results with comparable or lower token cost. In summary, VERISTRUCT solves substantially more benchmarks and verifies significantly more functions, demonstrating that our approach improves the capabilities of LLMs to generate correct annotations.

To provide a more fine-grained comparison, we inspect the verification trajectories of VERISTRUCT across all benchmarks. For each integer k , and for each benchmark, we compute the total number of functions fully verified within the first k LLM invocations. We then sum the numbers across all benchmarks for that value of k , plot the result, and then increase k . Plotting these points creates a visualization that shows the verification progress as a result of increasing LLM calls. We show the result in Fig. 7, where the x-axis denotes the LLM invocation index k , and the y-axis shows the cumulative number of functions verified within the first k invocations across all benchmarks.

Although the graph primarily trends upward, there are brief dips where the number of verified functions decreases. These temporary decreases occur during specification generation, when none of the generated specifications across the requested samples is syntactically correct. In such cases, VERISTRUCT must adopt a candidate specification that inadvertently breaks previously verified functionality. Our pipeline deliberately retains these specifications – even when they momentarily reduce coverage – so that subsequent repair modules can diagnose the faulty contracts once again and make progress towards verification. Typically, later stages recover from these dips and continue to progress toward full coverage.

We can observe the consistent dominance of VERISTRUCT over the baseline throughout the verification process. In particular, VERISTRUCT achieves the largest gain in verified functions during the second invocation (the first is for the planner), highlighting the effectiveness of our planner in selecting a highly effective initial module that verifies a large portion of functions early on. Overall, the experimental results clearly demonstrate that VERISTRUCT substantially outperforms the baseline, validating the effectiveness of our workflow design. Fig. 6 further shows the per-benchmark verification progression, where each benchmark makes steady progress toward its ground-truth target through iterative refinement.

Case Study: Bitmap. Interestingly, we find that the LLM produces a solution for the BITMAP benchmark that is fundamentally different from the ground-truth solution. In the ground-truth implementation written by human experts, the `View` trait models the bitmap as a two-dimensional array: each `u64` block is first abstracted into a bit sequence of length 64, and then the entire bitmap data structure is represented as a 2D array of these blocks. To manipulate this

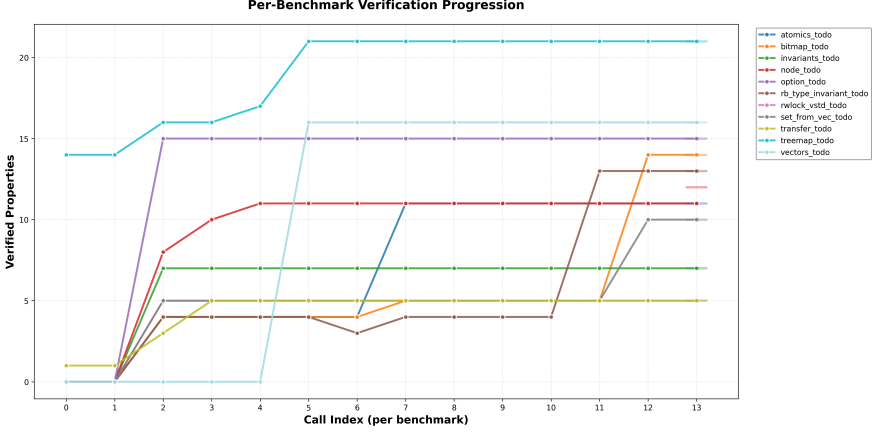


Fig. 6. Per-benchmark verification progression. Each benchmark makes steady progress toward its ground-truth target through iterative refinement, demonstrating consistent improvement across diverse verification tasks.

representation, the ground truth defines several auxiliary functions. In contrast, the LLM adopts a simpler abstraction—it models the entire bitmap as a single array, thereby eliminating the need for auxiliary functions and reasoning directly with Verus’ built-in APIs for the `Seq` type.

7 Related Work

Towards AI-assisted automated program verification, there is a long research line focusing on verifying the correctness of either an individual function [9, 20, 37–39] or the type invariant for a given module [29]. In contrast, this paper targets the verification of data structure modules, which requires the synthesis of mathematical abstractions, structural invariants, and specifications and proof blocks spanning multiple methods. Consequently, none of the previous work can be applied in our setting.

Traditional formal-methods tools (e.g., Frama-C [5], Why3 [7], Dafny [15], Coq [31], and Isabelle [21]) provide strong correctness guarantees but require developers to manually construct detailed specifications and proofs. Thus, they impose substantial annotation overhead and demand significant expertise. In contrast, VERISTRUCT leverages large language models to automatically synthesize annotations. As a result, VERISTRUCT bridges the gap between fully manual formal verification and prior LLM-based assistants, achieving greater automation without sacrificing rigor.

8 Conclusion and Future Work

This paper introduced VERISTRUCT, a novel framework that combines large language models with Verus to automatically verify Rust data-structure modules.

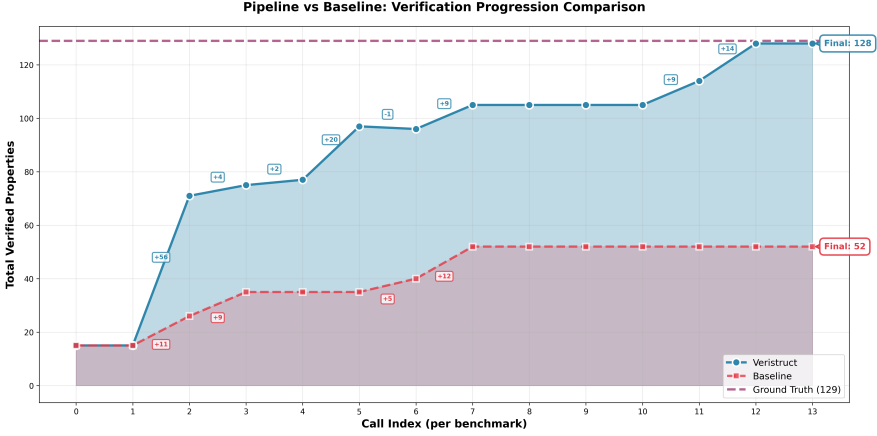


Fig. 7. Aggregated verification progress across all benchmarks. VERISTRUCT rapidly approaches the ground-truth total of 129 functions, while the single-shot baseline plateaus well below full coverage.

VERISTRUCT systematically generates **View** implementations, type invariants, specifications, and proof code under the scheduling of an extra planner module. To mitigate the issue that LLMs do not fully grasp Verus’ annotation syntax and verification-specific semantics, we not only inject syntax guidelines in the prompt, but also introduce an additional repair stage to fix errors.

We evaluated VERISTRUCT on eleven benchmarks drawn from the Verus GitHub repository. Across eleven benchmark modules, VERISTRUCT successfully verifies ten of them and verifies 128/129 (99.2%) of functions in our benchmark. VERISTRUCT shows a significant improvement over the baseline, demonstrating the framework’s effectiveness.

We conclude with several promising directions for future work. A natural next step is to extend VERISTRUCT to support more complex verification tasks. First, we could integrate retrieval-augmented generation (RAG) [17], which could help identify useful lemmas from the Verus standard library [14], strengthening proof synthesis. Second, we could additionally support the synthesis of Verus’ resource algebra library [12] for verifying complex concurrent data structures. Finally, applying constrained decoding techniques [8] could alleviate syntax errors in generated annotations.

Automatic Unit Test Generation. At present, our approach requires users to manually provide a comprehensive unit test suite. However, constructing such suites, especially those that achieve high coverage and include corner cases, can be both time-consuming and error-prone. Recent studies have shown that large language models are highly capable of generating high-quality test cases, including those targeting rare or edge conditions [18, 35]. Building on these advances, we plan to incorporate LLM-based unit test generation into our framework, thereby

Table 3. Detailed Statistics of VERISTRUCT, where #Funcs refers to the number of functions to be verified in each benchmark.

Benchmark	#Funcs	Time (minutes)	#LLM Calls	Solved?
ATOMICS	11	2.1	8	✓
BITMAP	14	12.2	13	✓
TREEMAP	21	0.3	6	✓
INVARIANTS	7	1.5	4	✓
NODE	12	10.4	12	✗ (11/12 Verified)
OPTION	15	2.6	3	✓
RINGBUFFER	13	4.2	12	✓
RWLOCKVSTD	5	6.4	3	✓
SETFROMVEC	10	9.1	13	✓
TRANSFER	5	1.2	4	✓
VECTORS	16	3.1	6	✓

further reducing manual effort and enhancing the completeness of the verification process.

Reinforcement Learning for Verification. Another promising direction is to apply reinforcement learning (RL) to specification inference and proof generation. The Verus verifier provides a natural reward signal: successful verification yields positive feedback, while failed verification attempts with diagnostic messages can guide policy updates. By training models to optimize for verification success, RL could enable more effective exploration of the specification and proof search space, potentially discovering annotation strategies that are difficult to obtain through prompting alone.

References

1. Artifact of veristruct (2025), <https://github.com/ChuyueSun/VeriStruct>
2. Anthropic: Claude code. <https://claude.ai/claude-code> (2025), accessed: 2025
3. Barrett, C., Boyd, B., Bursztein, E., Carlini, N., Chen, B., Choi, J., Chowdhury, A.R., Christodorescu, M., Datta, A., Feizi, S., Fisher, K., Hashimoto, T., Hendrycks, D., Jha, S., Kang, D., Kerschbaum, F., Mitchell, E., Mitchell, J., Ramzan, Z., Shams, K., Song, D., Taly, A., Yang, D.: Identifying and mitigating the security risks of generative ai. *Foundations and Trends in Privacy and Security* **6**(1), 1–52 (2023). <https://doi.org/10.1561/33000000041>, <http://dx.doi.org/10.1561/33000000041>
4. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) *Handbook of Satisfiability*, Second Edition, *Frontiers in Artificial Intelligence and Applications*, vol. 336, chap. 33, pp. 825–885. IOS Press (Feb 2021), <http://theory.stanford.edu/~barrett/pubs/BSST21.pdf>

5. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) *Proceedings of the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012)*. Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_16
6. Dong, Q., Li, L., Dai, D., Zheng, C., Ma, J., Li, R., Xia, H., Xu, J., Wu, Z., Liu, T., et al.: A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (2022)
7. Filiâtre, J.C., Paskevich, A.: Why3. <https://why3.lri.fr> (2024), version 1.6.0
8. Fu, Y., Baker, E., Ding, Y., Chen, Y.: Constrained decoding for secure code generation. *arXiv preprint arXiv:2405.00218* (2024)
9. Kamath, A., Senthilnathan, A., Chakraborty, S., Deligiannis, P., Lahiri, S.K., Lal, A., Rastogi, A., Roy, S., Sharma, R.: Finding inductive loop invariants using large language models. *arXiv preprint arXiv:2311.07948* (2023)
10. Knuth, D.E.: *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley (1997)
11. Lahiri, S.K.: Evaluating llm-driven user-intent formalization for verification-aware languages. In: *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2024*. p. 142 (2024)
12. Lattuada, A., Hance, T., Bosamiya, J., Brun, M., Cho, C., LeBlanc, H., Srinivasan, P., Achermann, R., Chajed, T., Hawblitzel, C., Howell, J., Lorch, J.R., Padon, O., Parno, B.: Verus: A practical foundation for systems verification. In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. p. 438–454. SOSP '24, Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3694715.3695952>, <https://doi.org/10.1145/3694715.3695952>
13. Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.: Verus: Verifying Rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages* (2023). <https://doi.org/10.1145/3586037>
14. Lattuada, A., Parno, B., Bosamiya, J., Hawblitzel, C., Hance, T., et al.: vstd: Verus standard library. <https://github.com/verus-lang/verus/tree/main/vstd> (Jun 2024), version 0.0.0, MIT License
15. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2010)*. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
16. Levy, A., Campbell, B., Ghena, B., Giffin, D.B., Pannuto, P., Dutta, P., Levis, P.: Multiprogramming a 64kb computer safely and efficiently. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. p. 234–251. SOSP '17, Association for Computing Machinery, New York, NY, USA (2017). <https://doi.org/10.1145/3132747.3132786>, <https://doi.org/10.1145/3132747.3132786>
17. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.t., Rocktäschel, T., Riedel, S., Kiela, D.: Retrieval-augmented generation for knowledge-intensive nlp tasks. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS '20, Curran Associates Inc., Red Hook, NY, USA (2020)
18. Liu, K., Chen, Z., Liu, Y., Zhang, J.M., Harman, M., Han, Y., Ma, Y., Dong, Y., Li, G., Huang, G.: Llm-powered test case generation for detecting bugs in plausible programs (2025), <https://arxiv.org/abs/2404.10304>

19. Miltner, A., Padhi, S., Millstein, T., Walker, D.: Data-driven inference of representation invariants. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1–15. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). <https://doi.org/10.1145/3385412.3385967>, <https://doi.org/10.1145/3385412.3385967>
20. Misu, M.R.H., Lopes, C.V., Ma, I., Noble, J.: Towards ai-assisted synthesis of verified Dafny methods. *Proc. ACM Softw. Eng.* **1**(FSE) (Jul 2024). <https://doi.org/10.1145/3643763>, <https://doi.org/10.1145/3643763>
21. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle. <https://isabelle.in.tum.de> (2024), version 2023
22. O’Hearn, P.W.: Separation logic. *Communications of the ACM* **62**(2), 86–95 (Feb 2019). <https://doi.org/10.1145/3211968>
23. OpenAI: Openai o1 system card. Technical report, OpenAI (2024), includes safety evaluations and red teaming results for the o1 and o1-mini models
24. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’05). pp. 247–258. Association for Computing Machinery, New York, NY, USA (2005). <https://doi.org/10.1145/1040305.1040327>
25. Perry, N., Srivastava, M., Kumar, D., Boneh, D.: Do users write more insecure code with AI assistants? In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 2785–2799. ACM (2023). <https://doi.org/10.1145/3576915.3623157>
26. Poesia, G., Loughridge, C., Amin, N.: dafny-annotator: AI-assisted verification of Dafny programs. *CoRR abs/2411.15143* (2024), <https://arxiv.org/abs/2411.15143>, arXiv:2411.15143
27. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002). pp. 55–74. IEEE Computer Society, Los Alamitos, CA, USA (2002). <https://doi.org/10.1109/LICS.2002.1029817>
28. Sun, C.E., Gao, S., Weng, T.W.: Breaking the barrier: Enhanced utility and robustness in smoothed drl agents. *ICML* (2024)
29. Sun, C., Agashe, V., Chakraborty, S., Taneja, J., Barrett, C.W., Dill, D.L., Qiu, X., Lahiri, S.K.: ClassInvGen: Class invariant synthesis using large language models. *CoRR abs/2502.18917* (2025), <https://arxiv.org/abs/2502.18917>
30. Sun, C., Sheng, Y., Padon, O., Barrett, C.: Clover: Closed-loop verifiable code generation. In: *AI Verification (SAIV 2024)*. Lecture Notes in Computer Science, vol. 14846, pp. 134–155. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-65112-0_7
31. The Coq Development Team: The Coq proof assistant. <https://coq.inria.fr> (2024), version 8.19.0
32. Verus Contributors: Verus tutorial and reference (2025), <https://verus-lang.github.io/verus/guide/>
33. Verus Contributors: vstd: Verus standard library api documentation (2025), <https://verus-lang.github.io/verus/verusdoc/vstd/>
34. Verus Project: Verilib. <https://verilib.org/>, accessed: 2025-02-17
35. Wang, Z., Liu, K., Li, G., Jin, Z.: Hits: High-coverage LLM-based unit test generation via method slicing. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. p. 1258–1268. ASE ’24, Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3691620.3695501>, <https://doi.org/10.1145/3691620.3695501>

36. Wang, Z., Zhou, Z., Song, D., Huang, Y., Chen, S., Ma, L., Zhang, T.: Towards understanding the characteristics of code generation errors made by large language models. In: Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE) (2025), to appear
37. Wen, C., Cao, J., Su, J., Xu, Z., Qin, S., He, M., Li, H., Cheung, S.C., Tian, C.: Enchanting program specification synthesis by large language models using static analysis and program verification. In: Computer Aided Verification (CAV 2024). Lecture Notes in Computer Science, vol. 14682, pp. 302–328. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_16
38. Wu, H., Barrett, C., Narodytska, N.: Lemur: Integrating large language models in automated program verification. In: The Twelfth International Conference on Learning Representations
39. Yang, C., Li, X., Misu, M.R.H., Yao, J., Cui, W., Gong, Y., Hawblitzel, C., Lahiri, S., Lorch, J.R., Lu, S., Yang, F., Zhou, Z., Lu, S.: AutoVerus: Automated proof generation for Rust code. CoRR **abs/2409.13082** (2024), <https://arxiv.org/abs/2409.13082>

Data Availability Statement. The source code of VERISTRUCT, including all prompts and the benchmark suite used in our evaluation, is publicly available at <https://github.com/ChuyueSun/VeriStruct>.

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Program Verification



On Deciding Constant Runtime of Linear Loops

Florian Frohn¹ , Jürgen Giesl¹ , Peter Giesl² , and Nils Lommen¹

¹ RWTH Aachen University, Aachen, Germany

² Department of Mathematics, University of Sussex, Brighton, UK

Abstract. We consider linear single-path loops of the form

while φ **do** $\vec{x} \leftarrow A\vec{x} + \vec{b}$ **end**

where $\vec{x}$ is a vector of variables, the loop guard φ is a conjunction of linear inequations over the variables $\vec{x}$, and the update of the loop is represented by the matrix A and the vector $\vec{b}$. It is already known that termination of such loops is decidable. In this work, we consider loops where A has real eigenvalues, and prove that it is decidable whether the loop's runtime (for all inputs) is bounded by a constant if the variables range over $\mathbb{R}$ or $\mathbb{Q}$. This is an important problem in automatic program verification, since safety of linear **while**-programs is decidable if all loops have constant runtime, and it is closely connected to the existence of multiphase-linear ranking functions, which are often used for termination and complexity analysis. To evaluate its practical applicability, we also present an implementation of our decision procedure.

1 Introduction

We investigate decidability of the question whether a linear³ single-path loop has constant runtime (i.e., whether the runtime of the loop can be bounded by the same constant for all inputs).

Example 1 (Leading Example). We consider linear single-path loops like

while $0 \leq x + y \leq 10$ **do** $\underbrace{\begin{pmatrix} x \\ y \end{pmatrix}}_{\vec{x}} \leftarrow \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}}_A \underbrace{\begin{pmatrix} x \\ y \end{pmatrix}}_{\vec{x}} + \underbrace{\begin{pmatrix} 1 \\ 0 \end{pmatrix}}_{\vec{b}}$ **end**

with linear arithmetic only, conjunctive loop guards, and without branching in the loop body. In our example, the loop body is evaluated if $0 \leq x + y \leq 10$. The value of x is increased by 1 and y is doubled in every iteration. Our goal is to decide whether the runtime of such loops is bounded by a constant. Later we will show that the loop in our example has indeed constant runtime.

Throughout this paper, the coefficients that occur in the loop are always rational numbers.⁴ Regarding the domain of the variables $\vec{x}$, we focus on loops over $\mathbb{R}$, but our results also apply to the case that the variables range over $\mathbb{Q}$.

³ In this paper, the term “linear” refers to linear *polynomials*, i.e., functions of the form $c_0 + \sum_{i=1}^d c_i \cdot x_i$ (in contrast to linear *transformations* of the form $\sum_{i=1}^d c_i \cdot x_i$).

⁴ Our approach also works if the coefficients are algebraic real numbers and the variables range over $\mathbb{R}$. We restrict ourselves to rational coefficients for simplicity.

The motivation for our work is three-fold. First, it is obviously relevant in the context of automated complexity analysis, in order to prove constant runtime. Second, the question whether a linear loop admits a *multiphase-linear ranking function* (M Φ RF) can be reduced to the question whether another linear loop has constant runtime w.r.t. a given set of initial states [5, Sect. 4.2]. Thus, our work is an important step towards a solution for the long-standing open problem whether the existence of M Φ RFs is decidable for linear loops. M Φ RFs are relevant for both automated termination and complexity analysis, since the existence of a M Φ RF does not only prove termination, but it also proves that the runtime of the loop is at most linear [4, Thm. 7]. However, our approach does not consider initial conditions, so it cannot be used for the inference of M Φ RFs yet. We intend to refine it to support initial conditions in future work. Third, our work is of interest for all infinite-state verification techniques that use approximations to deal with loops, like abstract interpretation, symbolic execution, recurrence analysis, loop summarization, etc. All of these techniques make use of approximations in the presence of loops to prevent divergence of the verification process, but at the price of losing precision. Instead, a loop with constant runtime c can simply be unrolled c times. Thus, reachability (and hence, safety) is decidable for all those linear **while**-programs where all loops have constant runtime.

The constant runtime problem turns out to be surprisingly challenging: Termination of linear loops is decidable over the reals [28], the rationals [7], and the integers [17]. But even for classes of loops where decidability of termination has been known for decades [28], decidability of the constant runtime problem is far from obvious. Intuitively, the reason is that all decidability results for termination of linear loops exploit *eventual monotonicity*: For loops with non-negative real eigenvalues, the variable values start to behave monotonically after finitely many loop iterations (where the loop guard is ignored), and negative eigenvalues can easily be eliminated. For the constant runtime problem, this kind of reasoning is not suitable, as one cannot disregard finitely many loop iterations.

Contributions: After introducing our notion of loops formally in Sect. 2, in Sect. 3 we prove decidability of the constant runtime problem for linear loops over $\mathbb{R}$ with real eigenvalues. Our decision procedure encodes the constant runtime problem as a validity problem, and then exploits the restriction to real eigenvalues to remove one quantifier alternation, so that the resulting formula is amenable to Fourier-Motzkin variable elimination. Our results immediately extend to loops over $\mathbb{Q}$, but not over $\mathbb{Z}$, due to the use of Fourier-Motzkin variable elimination. For loops over $\mathbb{Z}$, we show decidability of the constant runtime problem for linear loops with eigenvalues from $\{-1, 0, 1\}$ in Sect. 4. In Sect. 5, we discuss related work, and in Sect. 6 we evaluate our implementation and conclude.

2 Preliminaries

We consider loops of the form

$$\mathbf{while} \quad \varphi \quad \mathbf{do} \quad \vec{x} \leftarrow A\vec{x} + \vec{b} \quad \mathbf{end} \qquad (\text{LOOP})$$

where $\vec{x}$ is a vector of $d \geq 1$ variables, φ is a conjunction of linear inequations over $\vec{x}$ of the form $t \sim 0$ with $\sim \in \{>, \geq\}$, $A \in \mathbb{Q}^{d \times d}$, and $\vec{b} \in \mathbb{Q}^d$. We sometimes regard a conjunction of formulas $\varphi \equiv \varphi_1 \wedge \dots \wedge \varphi_m$ as a set $\{\varphi_1, \dots, \varphi_m\}$ such that we can write $\varphi_i \in \varphi$ for every conjunct φ_i . For any arithmetic expression t containing variables from $\vec{x}$, we define the *update function* up of **(LOOP)** as $\text{up}(t) := t[\vec{x}/A\vec{x} + \vec{b}]$, where $[\vec{x}/A\vec{x} + \vec{b}]$ denotes the substitution which replaces each variable from $\vec{x}$ by the corresponding polynomial from $A\vec{x} + \vec{b}$.

The loop **(LOOP)** has *constant runtime* if the following holds:

$$\exists c \in \mathbb{N}. \forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{\leq c}. \neg \varphi[\vec{x}/\text{up}^i(\vec{x})] \quad (\text{CONST})$$

Here, up^i denotes the i -fold application of up and $\mathbb{N}_{\leq c} = \{n \in \mathbb{N} \mid n \leq c\}$ for any $c \in \mathbb{N}$. Intuitively, **(CONST)** states that $c \in \mathbb{N}$ is a bound on the runtime of **(LOOP)** if, for every possible initial value $\vec{x} \in \mathbb{R}^d$, the loop guard is violated after $i \leq c$ iterations, i.e., $\neg \varphi[\vec{x}/\text{up}^i(\vec{x})]$ holds.

As in [10], we can restrict ourselves to *non-negative loops* (where A does not have negative eigenvalues) without loss of generality. The reason is that **(LOOP)** has constant runtime if and only if

$$\textbf{while } \varphi \wedge \varphi[\vec{x}/A\vec{x} + \vec{b}] \textbf{ do } \vec{x} \leftarrow A(A\vec{x} + \vec{b}) + \vec{b} \textbf{ end}$$

or, equivalently,

$$\textbf{while } \varphi \wedge \varphi[\vec{x}/A\vec{x} + \vec{b}] \textbf{ do } \vec{x} \leftarrow A^2\vec{x} + A\vec{b} + \vec{b} \textbf{ end} \quad (\text{NON-NEG})$$

has constant runtime. Here, the loop **(NON-NEG)** is obtained by *chaining* the loop **(LOOP)**, i.e., each iteration of **(NON-NEG)** corresponds to two subsequent iterations of **(LOOP)**. Clearly, if A only has real eigenvalues, then all eigenvalues of A^2 are non-negative. Now, Cor. 2 is a direct consequence of [14, Lemma 18].

Corollary 2 (Transformation to Non-Negative Loops). **(LOOP)** has constant runtime iff **(NON-NEG)** has constant runtime.

3 Loops with Real Eigenvalues

From now on, we only consider loops with real eigenvalues. Then we can restrict ourselves to loops like **(LOOP)** where all eigenvalues of A are non-negative without loss of generality due to Cor. 2. The advantage of loops like **(LOOP)** is that they admit suitable closed forms capturing the behavior of n loop iterations by a single expression for each variable. These closed forms are heavily used in complete techniques for termination analysis, invariant generation, and complexity analysis (see, e.g., [7, 10, 14, 15, 17–19, 25, 28]).

The remainder of this section is structured as follows: First, we introduce closed forms for linear loops with real eigenvalues in Sect. 3.1. Then, we use these closed forms to reduce the constant runtime problem to a validity problem in Sect. 3.2, i.e., from that point onward, our goal is to decide validity of a first-order formula **(CONST_{n0})** with a specific structure. The arithmetic expressions

in this formula contain the variables $\vec{x}$ from the loop under consideration, and an additional, specific variable that serves as a “loop counter”. Next, in Sect. 3.3 we show that the number of real roots of these arithmetic expressions w.r.t. the loop counter can be bounded by a constant that does not depend on the values of $\vec{x}$. In Sect. 3.4, we exploit this fact to eliminate the innermost quantifier from (CONST_{n_0}). For this step, it is crucial that all eigenvalues are non-negative real numbers. Without the innermost quantifier, the resulting formula is amenable to a variant of Fourier-Motzkin variable elimination that allows us to eliminate the variables $\vec{x}$, see Sect. 3.5 and 3.6. In this way, we obtain a univariate formula that contains both polynomial and exponential arithmetic. Finally, we show how to decide validity of such formulas in Sect. 3.7.

3.1 Closed Forms

Let $\mathbb{R}_{\mathbb{A}}$ denote the algebraic reals (i.e., $\mathbb{R}_{\mathbb{A}}$ contains every real number that is a root of a univariate, non-zero polynomial with integer coefficients). Moreover, $\mathbb{R}_{\mathbb{A}>0}$ denotes the positive algebraic reals. Similar to, e.g., [18, Thm. 3.1] and [15, Def. 3.4], for loops with non-negative real eigenvalues one can compute *poly-exponential* closed forms for the n -fold update of the loop. They have the form

$$\sum_{i=1}^M a_i(\vec{x}) \cdot b_i^n \cdot n^{e_i} \quad (\text{PE})$$

where $b_i \in \mathbb{R}_{\mathbb{A}>0}$, $e_i \in \mathbb{N}$, and $a_i \in \mathbb{R}_{\mathbb{A}}[\vec{x}]$ is a linear polynomial over the variables $\vec{x}$. Here, we identify a_i and the function $\vec{x} \mapsto a_i$ where we sometimes write $a_i(\vec{x})$ to make the variables $\vec{x}$ explicit, and we use the same convention for other (vectors of) expressions. Note that the b_i are the eigenvalues of the update matrix A . We highlight that the constants in (PE) are algebraic, as it is unclear how to compute with non-algebraic numbers, and thus closed forms involving such numbers would not be suitable for our decision procedure.

To compute such a closed form, the idea is to consider the Jordan normal form $J = \text{diag}(B_1, \dots, B_N)$ of the update matrix A , where $A = S \cdot J \cdot S^{-1}$ for a suitable change-of-basis matrix $S \in \mathbb{R}_{\mathbb{A}}^{d \times d}$. Here, each Jordan block B is associated to an eigenvalue $b \in \mathbb{R}_{\mathbb{A}>0}$. Note that the Jordan normal form (and all relevant computations regarding algebraic numbers) can always be computed in polynomial time [8]. Then the n -fold update is represented by the matrix $A^n = (S \cdot J \cdot S^{-1})^n = S \cdot J^n \cdot S^{-1}$ (i.e., S and S^{-1} always cancel out). Thus, one only has to compute a closed form for each Jordan block of $J^n = \text{diag}(B_1^n, \dots, B_N^n)$. For every Jordan block $B \in \mathbb{R}_{\mathbb{A}>0}^{d_B \times d_B}$, this closed form is

$$B^n = \begin{pmatrix} b^n \binom{n}{1} b^{n-1} & \dots & \binom{n}{d_B-1} \cdot b^{n-(d_B-1)} \\ 0 & b^n & \dots & \binom{n}{d_B-2} \cdot b^{n-(d_B-2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & b^n \end{pmatrix} \quad \text{for all } n \in \mathbb{N}$$

if $b \neq 0$, or $B^n = 0 \in \mathbb{R}_{\mathbb{A} \geq 0}^{d_B \times d_B}$ for all $n \geq d_B$ if $b = 0$. Note that the binomial coefficients yield the polynomial factors n^{e_i} of the poly-exponential expression (PE). Furthermore, in combination with the change-of-basis matrix $S \in \mathbb{R}_{\mathbb{A}}^{d \times d}$, the coefficients in the polynomials $a_i(\vec{x})$ are determined.

Hence, for any loop of the form (LOOP) with non-negative real eigenvalues only, there is a vector $\vec{cl}$ of d poly-exponential expressions of the form (PE) and a constant $n_0 \in \mathbb{N}$ such that

$$\text{up}^n(\vec{x}) = \vec{cl} \quad \text{for all } n \in \mathbb{N} \text{ with } n \geq n_0.$$

More precisely, n_0 is the dimension d_B of the largest Jordan block B which is associated to the eigenvalue $b = 0$.

Example 3 (Closed Forms, Ex. 1 cont.). For Ex. 1, we have the closed form

$$\vec{cl} = \begin{pmatrix} x+n \\ 2^n y \end{pmatrix}.$$

To see why the constant n_0 is needed for closed forms, consider the loop

$$\text{while } x \leq 0 \text{ do } \begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \text{ end}$$

Here, we have $n_0 = 2$, as the update matrix is already in Jordan normal form, its only eigenvalue is 0, and it has a single Jordan block of size 2. Then we obtain the closed form $\vec{cl} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. However, for $n \in \{0, 1\}$ we have

$$\vec{cl}[n/0] = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \neq \begin{pmatrix} x \\ y \end{pmatrix} = \text{up}^0(\vec{x}) \quad \text{and} \quad \vec{cl}[n/1] = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \neq \begin{pmatrix} y \\ 1 \end{pmatrix} = \text{up}^1(\vec{x}).$$

3.2 From Constant Runtime to Validity

We now reduce the constant runtime problem to a validity problem. As a first step, recall the characterization of constant loops:

$$\exists c \in \mathbb{N}. \forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{\leq c}. \neg \varphi[\vec{x}/\text{up}^i(\vec{x})] \quad (\text{CONST})$$

By “shifting” c by one, we obtain the equivalent formula

$$\exists c' \in \mathbb{N}. \forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{< c'}. \neg \varphi[\vec{x}/\text{up}^i(\vec{x})] \quad (\text{CONST}_{<})$$

which is more suitable for our further analysis (in particular in Sect. 3.3). Thus, if (CONST_<) holds, then $c' - 1$ is a bound on the runtime of (LOOP). We now show that (CONST_<) is equivalent to

$$\exists c \in \mathbb{N}. \forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{< c}. \neg \varphi[\vec{x}/\text{up}^{n_0+i}(\vec{x})]$$

where n_0 is defined as in Sect. 3.1 This allows us to directly replace $\text{up}^{n_0+i}(\vec{x})$ by its closed form, since $i \geq 0$. To this end, we define the *instantiated loop guard*

$$\psi := \varphi[\vec{x}/\vec{cl}].$$

So we show that (CONST_<) is equivalent to

$$\exists c \in \mathbb{N}. \forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{< c}. \neg \psi[n/n_0 + i], \quad (\text{CONST}_{n_0})$$

and if (CONST_{n₀}) holds, then $n_0 + c - 1$ is a bound on the runtime of (LOOP).

Proving Equivalence of $(\text{CONST}_{<})$ and (CONST_{n_0}) : To show that $(\text{CONST}_{<})$ and (CONST_{n_0}) are equivalent, we first prove that (CONST_{n_0}) implies $(\text{CONST}_{<})$. To this end, assume that (CONST_{n_0}) holds, but $(\text{CONST}_{<})$ does not. Hence, there exists a $c \in \mathbb{N}$ such that

$$\forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{< c}. \neg \psi[n/n_0 + i]. \quad (\text{CONST}_{n_0}^{in})$$

As $(\text{CONST}_{<})$ does not hold, for $c' = n_0 + c$ there is an $\vec{x} \in \mathbb{R}^d$ such that

$$\forall i \in \mathbb{N}_{< n_0 + c}. \varphi[\vec{x}/\text{up}^i(\vec{x})],$$

which contradicts $(\text{CONST}_{n_0}^{in})$, as for every $i \geq 0$ we have $\neg \psi[n/n_0 + i] \equiv \neg \varphi[\vec{x}/\text{up}^{n_0+i}(\vec{x})]$.

To see why $(\text{CONST}_{<})$ implies (CONST_{n_0}) , assume that $(\text{CONST}_{<})$ holds, but (CONST_{n_0}) does not. Hence, there exists a $c' \in \mathbb{N}$ such that

$$\forall \vec{y} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{< c'}. \neg \varphi[\vec{x}/\text{up}^i(\vec{y})]. \quad (\text{CONST}_{<}^{in})$$

As (CONST_{n_0}) does not hold, for $c = c'$ there exists an $\vec{x} \in \mathbb{R}^d$ such that

$$\forall i \in \mathbb{N}_{< c'}. \psi[n/n_0 + i]. \quad (\dagger)$$

Now we set $\vec{y} = \text{up}^{n_0}(\vec{x})$ in $(\text{CONST}_{<}^{in})$, i.e., there exists an $i \in \mathbb{N}_{< c'}$ such that $\neg \varphi[\vec{x}/\text{up}^i(\vec{y})]$, or equivalently, $\neg \varphi[\vec{x}/\text{up}^{n_0+i}(\vec{x})]$ holds. This is a contradiction to $(\dagger)$, as for every $i \geq 0$ we again have $\psi[n/n_0 + i] \equiv \varphi[\vec{x}/\text{up}^{n_0+i}(\vec{x})]$.

While $(\text{CONST}_{<})$ expresses that the loop guard must be violated within the first c' iterations, (CONST_{n_0}) “ignores” the first n_0 iterations. This allows us to use closed forms which are only valid for $n \geq n_0$.

Example 4 (Constant Runtime to Validity, Ex. 3 cont.). For Ex. 1, it remains to decide validity of

$$\exists c \in \mathbb{N}. \forall x, y \in \mathbb{R}. \exists i \in \mathbb{N}_{< c}. \neg (0 \leq x + i + 2^i y \leq 10). \quad (\text{VAL})$$

Moreover, if (VAL) holds, then $c - 1$ is a bound on the runtime of Ex. 1.

3.3 Bounded Number of Real Roots

Recall that (CONST_{n_0}) states that there is a $c \in \mathbb{N}$ such that for all initial values $\vec{x}$, there is an iteration n with $n_0 \leq n < n_0 + c$ where the loop guard does not hold. So here, we have to consider all instantiations of n with natural numbers from n_0 up to $n_0 + c - 1$. For deciding validity of (CONST_{n_0}) , the problem is that c is not yet known. Our aim is to replace (CONST_{n_0}) by

$$\exists m \in \mathbb{N}. \forall \vec{x} \in \mathbb{R}^d. \bigvee_{j=0}^{rb} \neg \psi[n/n_0 + j \cdot m], \quad (\text{QE})$$

where we only check $rb + 1$ instantiations of n . Here, rb is a number that only depends on ψ , but not on the actual instantiation of $\vec{x}$. Thus, our goal is to replace

the existentially quantified formula $\exists i \in \mathbb{N}_{<c}. \neg \psi[n/n_0 + i]$ which depends on the (unknown) value c by the disjunction $\bigvee_{j=0}^{rb} \neg \psi[n/n_0 + j \cdot m]$, where rb is known and fixed. So to eliminate the inner existential quantifier in (CONST_{n_0}) , we show that it suffices to consider a fixed number of “sample points” instead of all values $i \in \mathbb{N}_{<c}$. Here, the (existentially quantified) variable m is used as the distance between our “sample points”.

To determine the number rb , n is instantiated by real instead of just natural numbers. Then, the number rb is chosen in such a way that for every vector $\vec{v}$ of real numbers, the expressions in the inequations of the formula $\psi[\vec{x}/\vec{v}]$ (which only contains the variable n) have at most rb real roots. Note that the number of real roots is unbounded without the restriction to non-negative real eigenvalues.

To define rb , for any poly-exponential expression t as in (PE), we first define its **rootbound**. Here, **rootbound**(t) only depends on the number M of addends and the exponents e_i , but **rootbound**(t) does not depend on $\vec{x}$. The following lemma shows that for every value $\vec{v} \in \mathbb{R}^d$, **rootbound**(t) is an upper bound on the number of real roots of the expression $t[\vec{x}/\vec{v}]$.

Lemma 5 (Bounded Number of Real Roots). *Let $\vec{v} \in \mathbb{R}^d$ and let*

$$t = \sum_{i=1}^M a_i(\vec{x}) \cdot b_i^n \cdot n^{e_i}$$

be a poly-exponential expression such that $t[\vec{x}/\vec{v}]$ is not the polynomial 0. Then

$$\text{rootbound}(t) := M - 1 + \sum_{i=1}^M e_i$$

is an upper bound on the number of real roots of $t[\vec{x}/\vec{v}]$.

Proof. As in [26], for any expression s of the form

$$\sum_{i=1}^{M'} p_i(n) \cdot b_i^n, \tag{1}$$

where each b_i is unique and each $p_i(n)$ is a polynomial over n , we define

$$\text{ord}(s) := \sum_{i=1}^{M'} (\text{degree}(p_i(n)) + 1) = M' + \sum_{i=1}^{M'} \text{degree}(p_i(n)). \tag{2}$$

Let $\vec{v} \in \mathbb{R}^d$ be arbitrary but fixed. Then we can rewrite

$$t' = t[\vec{x}/\vec{v}] = \sum_{i=1}^M a_i(\vec{v}) \cdot b_i^n \cdot n^{e_i}$$

to the form (1), and thus by [26, Part V, Chap. 1, Num. 75], it has at most $\text{ord}(t') - 1$ real roots (where t' contains no other variable than n). As $M' \leq M$ and

$\sum_{i=1}^{M'} \text{degree}(p_i(n)) \leq \sum_{i=1}^M e_i$, this implies that t' has at most $M - 1 + \sum_{i=1}^M e_i$ real roots due to (2). As this bound is independent from $\vec{v}$, the claim follows. $\square$

We can now define the desired number rb .

Definition 6 (Root Bound of Loop Guard). *We define the root bound of the instantiated loop guard ψ as*

$$rb := \sum_{(t \sim 0) \in \psi} \text{rootbound}(t),$$

where $\sim \in \{>, \geq\}$, i.e., here we consider all inequations of the form $t > 0$ and $t \geq 0$ occurring in the conjunction ψ .

Example 7 (Root Bound, Ex. 4 cont.). For Ex. 1, $\psi = \varphi[\vec{x}/\vec{cl}]$ is

$$0 \leq x + n + 2^n y \leq 10 \quad \equiv \quad x + n + 2^n y \geq 0 \wedge 10 - x - n - 2^n y \geq 0.$$

Here, we have

$$\text{rootbound}(x + n + 2^n y) = \text{rootbound}(10 - x - n - 2^n y) = 3$$

since $M = 3$ and $\sum_{i=1}^3 e_i = 1$ and thus, $rb = 3 + 3 = 6$.

3.4 Eliminating the Innermost Quantifier

Now we show that (CONST_{n_0}) is equivalent to

$$\exists m \in \mathbb{N}. \forall \vec{x} \in \mathbb{R}^d. \bigvee_{j=0}^{rb} \neg \psi[n/n_0 + j \cdot m], \quad (\text{QE})$$

and if (QE) holds, then $n_0 + rb \cdot m$ is a bound on the runtime of (LOOP), again provided that all eigenvalues are non-negative reals.

Recall that (QE) states that there is a distance m such that for all initial values of $\vec{x}$, the guard is violated after $n_0, n_0 + 1 \cdot m, \dots$, or $n_0 + rb \cdot m$ iterations.

Proving Equivalence of (CONST_{n_0}) and (QE): To see why (CONST_{n_0}) and (QE) are equivalent, note that

$$(\text{QE}) \text{ implies } (\text{CONST}_{n_0}) \text{ by choosing } c = rb \cdot m + 1. \quad (\text{QE} \Rightarrow \text{CONST}_{n_0})$$

Thus, $n_0 + c - 1 = n_0 + rb \cdot m$ is a bound on the runtime of (LOOP).

To show that (CONST_{n_0}) implies (QE), assume (CONST_{n_0}) , i.e., there is a $c \in \mathbb{N}$ so that $\forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{<c}. \neg \psi[n/n_0 + i]$. Then for every $j \in \mathbb{N}$, we have:

$$\begin{aligned} & \forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{<c}. \neg \psi[n/n_0 + i] && (\text{for some } c \in \mathbb{N} \text{ by } (\text{CONST}_{n_0})) \\ \iff & \forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{<c}. \neg \varphi[\vec{x}/\vec{cl}][n/n_0 + i] && (\text{def. of } \psi) \end{aligned}$$

$$\begin{aligned}
&\iff \forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{<c}. \neg \varphi[\vec{x}/\vec{cl}[n/n_0 + i]] && (\text{as } n \text{ does not occur in } \varphi) \\
&\iff \forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{<c}. \neg \varphi[\vec{x}/\text{up}^{n_0+i}(\vec{x})] && (\text{def. of } \vec{cl}, \text{ as } i \geq 0) \\
&\implies \forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{<c}. \neg \varphi[\vec{x}/\text{up}^{n_0+j \cdot c+i}(\vec{x})] && (\text{for any } j \in \mathbb{N} (\dagger)) \\
&\iff \forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{<c}. \neg \varphi[\vec{x}/\vec{cl}[n/n_0 + j \cdot c + i]] && (\text{def. of } \vec{cl}, \text{ as } j \cdot c + i \geq 0) \\
&\iff \forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{<c}. \neg \varphi[\vec{x}/\vec{cl}][n/n_0 + j \cdot c + i] && (\text{as } n \text{ does not occur in } \varphi) \\
&\iff \forall \vec{x} \in \mathbb{R}^d. \exists i \in \mathbb{N}_{<c}. \neg \psi[n/n_0 + j \cdot c + i] && (\text{def. of } \psi)
\end{aligned}$$

For the step $(\dagger)$, $\vec{x}$ is again universally quantified, i.e., we can instantiate $\vec{x}$ by $\text{up}^{j \cdot c}(\vec{x})$, as in the proof that $(\text{CONST}_{<})$ implies (CONST_{n_0}) , where we instantiated the universally quantified variables $\vec{y}$ by $\text{up}^{n_0}(\vec{x})$. Thus, (CONST_{n_0}) implies

$$\forall \vec{x} \in \mathbb{R}^d. \bigwedge_{j=0}^{rb} \exists i \in \mathbb{N}_{<c}. \neg \psi[n/n_0 + j \cdot c + i]. \quad (\text{ININTERVALS})$$

So (ININTERVALS) holds iff for all initial values of $\vec{x}$ and all $0 \leq j \leq rb$, the loop guard is violated after $n_0 + j \cdot c$, $n_0 + j \cdot c + 1$, $\dots$, or $n_0 + (j+1) \cdot c - 1$ iterations.

To prove that (ININTERVALS) implies (QE) , we assume that (ININTERVALS) holds and (QE) does not hold, and derive a contradiction. If (QE) does not hold, then there is a $\vec{v} \in \mathbb{R}^d$ such that

$$\bigwedge_{j=0}^{rb} \psi[n/n_0 + j \cdot c][\vec{x}/\vec{v}] \quad (\text{ATBORDERS})$$

by choosing $m = c$ in (QE) . Then by the intermediate value theorem, for each $0 \leq j \leq rb$, there is at least one $(t_j \sim_j 0) \in \psi$ with $\sim_j \in \{>, \geq\}$ such that $t_j[\vec{x}/\vec{v}]$ has a real root in the interval

$$[n_0 + j \cdot c, n_0 + (j+1) \cdot c - 1] \quad (3)$$

due to (ININTERVALS) , if we consider instantiations of n with real instead of just natural numbers. The reason is that (ATBORDERS) implies $t[\vec{x}/\vec{v}][n/n_0 + j \cdot c] \sim 0$ for all $(t \sim 0) \in \psi$, and (ININTERVALS) implies that there exists a value $k \in [n_0 + j \cdot c, n_0 + (j+1) \cdot c - 1]$ and a $(t_j \sim_j 0) \in \psi$ such that $t_j[\vec{x}/\vec{v}][n/k] \sim_j 0$ is violated. Then we must also have $t_j[\vec{x}/\vec{v}][n/k_0] = 0 \neq t_j[\vec{x}/\vec{v}][n/k_{\neq 0}]$ for some $k_0, k_{\neq 0} \in [n_0 + j \cdot c, n_0 + (j+1) \cdot c - 1]$, since $t_j[\vec{x}/\vec{v}]$ is a continuous function in the variable n .⁵ In other words, for every interval (3) there is a $t_j[\vec{x}/\vec{v}]$ that satisfies $t_j[\vec{x}/\vec{v}] \sim_j 0$ at the “borders” of the interval and violates $t_j[\vec{x}/\vec{v}] \sim_j 0$ inside the interval. Thus, we have $t_j[\vec{x}/\vec{v}] \neq 0$ (i.e., $t_j[\vec{x}/\vec{v}]$ is not the polynomial 0), and it must have a real root in the interval. Hence, the sum of the number of real roots of all non-zero expressions $t[\vec{x}/\vec{v}]$, $(t \sim 0) \in \psi$, is at least $rb + 1$, i.e.,

$$\sum_{\substack{(t \sim 0) \in \psi \\ t[\vec{x}/\vec{v}] \neq 0}} |\text{roots}(t[\vec{x}/\vec{v}])| > rb = \sum_{(t \sim 0) \in \psi} \text{rootbound}(t).$$

⁵ The reason for continuity of $t_j[\vec{x}/\vec{v}]$ is that all exponential expressions in $t_j[\vec{x}/\vec{v}]$ have the form b_i^n for $b_i \in \mathbb{R}_{\geq 0}$, since all eigenvalues of the loop are non-negative reals.

This contradicts the fact that $\text{rootbound}(t)$ is a bound on the number of real roots of $t[\vec{x}/\vec{v}]$ by Lemma 5. Thus, as we instantiated m with c to derive (ATBORDERS), we showed

$\neg(\text{QE})$ and (CONST_{n_0}) yield a contradiction by choosing $m = c$

or, equivalently,

(CONST_{n_0}) implies (QE) by choosing $m = c$. $(\text{CONST}_{n_0} \Rightarrow \text{QE})$

Example 8 (Quantifier Elimination, Ex. 4 and 7 cont.). For (VAL), it remains to decide validity of

$$\exists m \in \mathbb{N}. \forall x, y \in \mathbb{R}. \bigvee_{j=0}^6 \neg (0 \leq x + j \cdot m + 2^{j \cdot m} y \leq 10). \quad (\text{VAL}^\vee)$$

In our example we have $n_0 = 0$ and $rb = 6$. Thus, if $(\text{VAL}^\vee)$ holds, then this yields the bound $n_0 + rb \cdot m = 6 \cdot m$ on the runtime of the loop in Ex. 1.

The proof above yields the following corollary, which will be needed later.

Corollary 9. *Consider the subformula*

$$\forall \vec{x} \in \mathbb{R}^d. \bigvee_{j=0}^{rb} \neg \psi[n/n_0 + j \cdot m] \quad (\text{QE}^{in})$$

of (QE), whose only free variable m ranges over $\mathbb{N}$. Then (QE^{in}) is either unsatisfiable, or it is valid for infinitely many different values of m .

Proof. First assume $rb = 0$. Then (QE^{in}) does not depend on m , and thus it is either unsatisfiable, or it is valid for *all* possible values of m .

Now let $rb > 0$ and assume that (QE^{in}) is only valid for finitely many values of m . Let $v \in \mathbb{N}$ be the maximal number such that instantiating m by v satisfies (QE^{in}) . Then with $(\text{QE} \Rightarrow \text{CONST}_{n_0})$, it follows that instantiating c with $rb \cdot v + 1$ satisfies $(\text{CONST}_{n_0}^{in})$. Next, by $(\text{CONST}_{n_0} \Rightarrow \text{QE})$, it follows that instantiating m with $rb \cdot v + 1$ satisfies (QE^{in}) . As $rb > 0$, we have $rb \cdot v + 1 > v$, i.e., (QE^{in}) also holds for a number larger than v , which contradicts our assumption on the maximality of v . $\square$

3.5 Eventual Signs

Compared to the formula (CONST_{n_0}) , in (QE) we eliminated the innermost quantifier and replaced it by $rb + 1$ subformulas, where rb is fixed. The resulting formula (QE) is linear in the program variables $\vec{x}$ and poly-exponential in m .

Next, we want to get rid of $\vec{x}$ by using a variant of Fourier-Motzkin elimination. To see why Fourier-Motzkin elimination can be applied even though we have

non-linear terms in m , we disregard the outermost quantifier of (QE) for now, i.e., we assume that $m \in \mathbb{N}$ is given. Then proving (QE) amounts to proving the *validity* of a *disjunction* of inequations. In contrast, Fourier-Motzkin elimination is used to check the *satisfiability* of *conjunctions*. Thus, to prove validity of (QE), our goal is to use Fourier-Motzkin elimination to prove unsatisfiability of

$$\pi := \bigwedge_{j=0}^{rb} \psi[n/n_0 + j \cdot m],$$

which results from the negation of (QE).

However, Fourier-Motzkin's technique applies to linear inequations only. To apply it in our setting, we regard the (poly-exponential) inequations in π as linear inequations over $\vec{x}$ whose coefficients are univariate poly-exponential expressions w.r.t. m , i.e., the coefficients are of the form

$$0 \quad \text{or} \quad \sum_{i=1}^M d_i \cdot b_i^m \cdot m^{e_i} \quad (\text{COEFF})$$

where m is the only variable, $M \geq 1$, $d_i \in \mathbb{R}_{\mathbb{A}} \setminus \{0\}$, $b_i \in \mathbb{R}_{\mathbb{A}>0}$, and $e_i \in \mathbb{N}$. Again, note that d_i and b_i are algebraic numbers, and thus they are suitable for algorithmic purposes. To apply Fourier-Motzkin elimination, the signs of these coefficients must be known, which is not the case, in general.

However, we can easily identify the asymptotically dominant addend of the expression (COEFF), which determines its sign *for large enough values of m* . Moreover, by Cor. 9, we may assume that m is sufficiently large without loss of generality: If (QE) holds for some small value of m , then there is also an arbitrarily large value of m where (QE) holds.

More formally, to determine the sign for sufficiently large m , w.l.o.g. assume that the pair (b_i, e_i) is unique for each addend in (COEFF). Then it suffices to consider the *asymptotically dominant* addend $d_j \cdot b_j^m \cdot m^{e_j}$ where $(b_j, e_j) \geq_{lex} (b_i, e_i)$ for all $1 \leq i \leq M$, i.e., (b_j, e_j) is maximal w.r.t. the lexicographic ordering. Since all eigenvalues b_i are non-negative reals, for sufficiently large m , the sign of the coefficient (COEFF) is equal to the sign of d_j or the coefficient is always 0. More precisely, for any number v , let:

$$\text{sign}(v) = -1 \text{ if } v < 0 \quad \text{sign}(0) = 0 \quad \text{sign}(v) = 1 \text{ if } v > 0$$

Note that since we required $d_i \neq 0$ and uniqueness of (b_i, e_i) for each addend, when instantiating m by large enough numbers, then we have $\text{sign}(\text{COEFF}) \neq 0$ whenever (COEFF) is of the form $\sum_{i=1}^M d_i \cdot b_i^m \cdot m^{e_i}$. In this case, we have

$$\exists m_0 \in \mathbb{N}. \forall m \geq m_0. \text{sign}(\text{COEFF}) = \text{sign}(d_j).$$

We define the *eventual sign* of (COEFF) as $\text{esign}(\text{COEFF}) := 0$ if (COEFF) is 0 and as $\text{esign}(\text{COEFF}) := \text{sign}(d_j)$ otherwise.

Example 10 (Eventual Signs, Ex. 8 cont.). Reconsider $(\text{VAL}^\vee)$. Here, the coefficients for the variable x are 1 and -1 , and we obviously have $\text{esign}(1) = 1$ and $\text{esign}(-1) = -1$. The coefficients for the variable y are $2^{j \cdot m}$ and $-2^{j \cdot m}$ for $0 \leq j \leq 6$, where $\text{esign}(2^{j \cdot m}) = 1$ and $\text{esign}(-2^{j \cdot m}) = -1$. For a more complex example, consider $(3^m \cdot m^2 - 10 \cdot 2^m \cdot m^3) \cdot x$. As $(3, 2) \geq_{lex} (2, 3)$, we have $\text{esign}(3^m \cdot m^2 - 10 \cdot 2^m \cdot m^3) = 1$, i.e., the asymptotically dominant coefficient of x eventually becomes positive.

3.6 Variable Elimination

We now show how to eliminate a single variable x in the vector $\vec{x}$ of variables from π . Then it immediately follows that *all* variables from $\vec{x}$ can be eliminated, so that the only remaining variable is m . Recall that π is defined as

$$\pi := \bigwedge_{j=0}^{rb} \psi[n/n_0 + j \cdot m].$$

So in particular, π is a conjunction of inequations. To eliminate x , consider the sets

$$\tilde{\pi}_x := \{p \cdot x + t \sim 0 \text{ occurs in } \pi \mid \sim \in \{>, \geq\}, \text{esign}(p) > 0\}$$

$$\hat{\pi}_x := \{p \cdot x + t \sim 0 \text{ occurs in } \pi \mid \sim \in \{>, \geq\}, \text{esign}(p) < 0\}$$

where p is the coefficient of x when regarding $p \cdot x + t$ as a linear polynomial w.r.t. $\vec{x}$ with coefficients of the form (COEFF) . In other words, p is chosen such that t does not contain any occurrence of x . Thus, $\tilde{\pi}_x$ contains those inequations from π that give rise to lower bounds on x for large enough m , and $\hat{\pi}_x$ contains those inequations that give rise to upper bounds. Next, we define π_x to be the smallest set such that the following holds:

$$\begin{array}{ll} \text{If } p_1 \cdot x + t_1 \sim_1 0 \in \tilde{\pi}_x & \text{and} \quad p_2 \cdot x + t_2 \sim_2 0 \in \hat{\pi}_x, \\ \text{then } p_1 \cdot t_2 - p_2 \cdot t_1 \sim 0 \in \pi_x, \end{array}$$

where $\sim$ is $\geq$ if both $\sim_1$ and $\sim_2$ are $\geq$, and $\sim$ is $>$, otherwise. Then

$$\pi \equiv \pi' \text{ for all large enough values of } m,$$

where $\pi' := (\pi \cup \pi_x) \setminus (\tilde{\pi}_x \cup \hat{\pi}_x)$ and by construction, x does not occur in π' .

Example 11 (Fourier-Motzkin Elimination, Ex. 8 and 10 cont.). For the formula $(\text{VAL}^\vee)$, we get the following inequations:

$$\begin{array}{ll} 0 \leq \overbrace{1}^p \cdot x + \overbrace{j \cdot m + 2^{j \cdot m} y}^t & (\text{for all } 0 \leq j \leq 6) \\ x + j \cdot m + 2^{j \cdot m} y \leq 10 & (\text{for all } 0 \leq j \leq 6) \end{array}$$

Thus, we have:

$$\begin{array}{l} \tilde{\pi}_x = \{x + j \cdot m + 2^{j \cdot m} y \geq 0 \mid 0 \leq j \leq 6\} \\ \hat{\pi}_x = \{-x - j \cdot m - 2^{j \cdot m} y + 10 \geq 0 \mid 0 \leq j \leq 6\} \end{array}$$

Hence, eliminating x yields

$$\begin{aligned}
\pi' &= \{j \cdot m + 2^{j \cdot m} y \geq j' \cdot m + 2^{j' \cdot m} y - 10 \mid j, j' \in \{0, \dots, 6\}\} \\
&= \{(j - j') \cdot m + (2^{j \cdot m} - 2^{j' \cdot m}) \cdot y + 10 \geq 0 \mid j, j' \in \{0, \dots, 6\}\} \\
&= \{(j - j') \cdot m + ((2^j)^m - (2^{j'})^m) \cdot y + 10 \geq 0 \mid j, j' \in \{0, \dots, 6\}\} \quad (\star)
\end{aligned}$$

Next, we eliminate y from π' , where we get:

$$\begin{aligned}
\tilde{\pi}'_y &\supseteq \{(4^m - 1) \cdot y + 2m + 10 \geq 0\} & ((\star) \text{ with } j = 2 \text{ and } j' = 0) \\
\hat{\pi}'_y &\supseteq \{(1 - 2^m) \cdot y - m + 10 \geq 0\} & ((\star) \text{ with } j = 0 \text{ and } j' = 1)
\end{aligned}$$

Thus, the resulting formula π'' contains the conjunct

$$\begin{aligned}
&(4^m - 1) \cdot (10 - m) - (1 - 2^m) \cdot (2m + 10) \geq 0 \\
&\equiv -4^m \cdot m + 10 \cdot 4^m + 2^m \cdot 2m + 10 \cdot 2^m - m - 20 \geq 0. \quad (\dagger)
\end{aligned}$$

Correctness of Variable Elimination: To see why this construction is correct, we show that

$$\pi \equiv \pi' \text{ for all large enough values of } m,$$

where $\pi' := (\pi \cup \pi_x) \setminus (\tilde{\pi}_x \cup \hat{\pi}_x)$. More precisely, we prove that $\pi[m/c]$ is equivalent to $\pi'[m/c]$ for every $c \in \mathbb{N}$ where $\text{sign}(p[m/c]) = \text{esign}(p)$ for all coefficients p in π .

Thus, if $m_0 \in \mathbb{N}$ such that for all $m \geq m_0$ we have $\text{sign}(p) = \text{esign}(p)$, then π is also equivalent to π' for all $m \geq m_0$. Hence, instead of checking unsatisfiability of π for some sufficiently large value of m , we can instead check unsatisfiability of π' for some sufficiently large value of m .

Equivalence of $\pi[m/c]$ and $\pi'[m/c]$: To show that $\pi[m/c]$ is equivalent to $\pi'[m/c]$ if $\text{sign}(p[m/c]) = \text{esign}(p)$ for all coefficients p in π , we can use the idea of Fourier-Motzkin elimination, since $\pi[m/c]$ is linear:

If all inequations are weak (i.e., built with $\geq$), then the inequations

$$p_1[m/c] \cdot x + t_1[m/c] \geq 0$$

in $\tilde{\pi}_x[m/c]$ are equivalent to $x \geq \frac{-t_1[m/c]}{p_1[m/c]}$ (since $p_1[m/c] > 0$ by definition of $\tilde{\pi}_x$). Similarly, the inequations

$$p_2[m/c] \cdot x + t_2[m/c] \geq 0$$

in $\hat{\pi}_x[m/c]$ are equivalent to $x \leq \frac{-t_2[m/c]}{p_2[m/c]}$ (since $p_2[m/c] < 0$ by definition of $\hat{\pi}_x$).

Hence, $\pi[m/c]$ is equivalent to the conjunction of $(\pi \setminus (\tilde{\pi}_x \cup \hat{\pi}_x))[m/c]$ and the inequations

$$\frac{-t_1[m/c]}{p_1[m/c]} \leq \frac{-t_2[m/c]}{p_2[m/c]} \quad (\text{FM})$$

for all $p_1 \cdot x + t_1 \geq 0$ from $\tilde{\pi}_x$ and all $p_2 \cdot x + t_2 \geq 0$ from $\hat{\pi}_x$.⁶

⁶ This does not hold for discrete sets like $\mathbb{Z}$, which is the reason why the approach of Sect. 3.6 cannot be applied for loops over $\mathbb{Z}$.

Note that if at least one of the two inequations $p_1 \cdot x + t_1 \geq 0$ or $p_2 \cdot x + t_2 \geq 0$ is strict, then (FM) is also strict. Since $p_1[m/c] > 0$ and $p_2[m/c] < 0$, we get

$$(FM) \equiv -p_2[m/c] \cdot t_1[m/c] \geq -p_1[m/c] \cdot t_2[m/c] \equiv (p_1 \cdot t_2 - p_2 \cdot t_1 \geq 0)[m/c].$$

Thus, $\pi[m/c]$ is equivalent to $(\pi \setminus (\tilde{\pi}_x \cup \hat{\pi}_x))[m/c] \cup \pi_x[m/c] = \pi'[m/c]$.

3.7 Deciding Validity

After eliminating all variables, we obtain a conjunction of inequations of the form $p > 0$ or $p \geq 0$ where p is of the form (COEFF). Such a conjunction is unsatisfiable for large enough values of m iff there is an inequation $p > 0$ or $p \geq 0$ where $\text{esign}(p) = -1$ or an inequation $p > 0$ with $\text{esign}(p) = 0$. Thus, for the case where $\text{esign}(p) = 0$, we have to distinguish between “ $>$ ” and “ $\geq$ ”.

However, checking satisfiability in this way does not directly yield an instantiation of m where π is unsatisfiable. Thus, we cannot directly obtain a corresponding constant bound $n_0 + rb \cdot m$ on the runtime. However, if π is proven to be unsatisfiable, then to compute the constant bound on the runtime of the analyzed loop, the loop can simply be unrolled until its guard becomes unsatisfiable.

Example 12 (Deciding Validity, Ex. 11 finished). For the formula ($\frac{1}{2}$), we have $\text{esign}(\frac{1}{2}) = -1$, as $-4^m \cdot m$ is the dominant addend, since we have:

$$\underbrace{-4^m \cdot m}_{(4,1)} >_{lex} \underbrace{10 \cdot 4^m}_{(4,0)} >_{lex} \underbrace{2^m \cdot 2m}_{(2,1)} >_{lex} \underbrace{10 \cdot 2^m}_{(2,0)} >_{lex} \underbrace{-m}_{(1,1)} >_{lex} \underbrace{-20}_{(1,0)}$$

Thus, ($\frac{1}{2}$) is unsatisfiable for sufficiently large values of m , and hence (VAL^V) is valid. Indeed, ($\frac{1}{2}$) is contradictory for all $m \geq 11$ (but not for $0 \leq m \leq 10$). Therefore, Ex. 1 has constant runtime, and we obtain the bound $n_0 + rb \cdot m = 0 + 6 \cdot 11 = 66$, i.e., every run takes at most 66 iterations.

So in this section, we derived the following theorem.

Theorem 13 (Constant Runtime over $\mathbb{R}$). *Constant runtime of linear loops over $\mathbb{R}$ with real eigenvalues is decidable.*

Moreover, our approach can also be applied to loops over $\mathbb{Q}$ without adaptations, as shown by the following lemma.

Lemma 14 (Constant over $\mathbb{R} \iff$ Constant over $\mathbb{Q}$). *A linear loop with real eigenvalues has constant runtime over $\mathbb{R}$ iff it has constant runtime over $\mathbb{Q}$.*

Proof. Constant runtime over the reals obviously implies constant runtime over the rationals. For the other direction, assume that the runtime over the rationals is bounded by the constant c . Then $\varphi^{(0..c)} := \varphi \wedge \text{up}(\varphi) \wedge \dots \wedge \text{up}^c(\varphi)$ is unsatisfiable over the rationals. Note that this formula only contains linear arithmetic. As formulas with linear arithmetic are satisfiable over the reals iff they are satisfiable over the rationals, $\varphi^{(0..c)}$ is also unsatisfiable over the reals. Hence, the runtime over the reals is also bounded by the constant c . $\square$

Then with Thm. 13, the following corollary is immediate.

Corollary 15 (Constant Runtime over $\mathbb{Q}$). *Constant runtime of linear loops over $\mathbb{Q}$ with real eigenvalues is decidable.*

4 Loops over $\mathbb{Z}$

We now consider loops over the integers, i.e., where the coefficients in (LOOP) are from $\mathbb{Z}$, and where the variables range over $\mathbb{Z}$, too. It is easy to see that all parts of the procedure presented in Sect. 3 immediately carry over to the integer setting except for one: We cannot eliminate variables via Fourier-Motzkin’s technique as in Sect. 3.6. However, instead we can exploit decidability of “one-parametric Presburger arithmetic” [6, 23]. This logic extends Presburger arithmetic with a single function $x \mapsto x \cdot m$, where m is a free variable. So the resulting formulas are non-linear polynomials w.r.t. m , but linear w.r.t. all other variables. Thus, we can apply decidability of this logic to (QE), but not directly to (CONST).

In our setting, recall that the bases b_i of the exponential functions that occur in closed forms correspond to the eigenvalues of the loop. Thus, if we restrict our attention to the eigenvalues 0 and 1, then all exponential functions vanish. Moreover, if we also allow the eigenvalue -1 , then after applying the simplification from Cor. 2, the only remaining eigenvalues are again 0 and 1. Thus, we obtain the following corollary.

Corollary 16 (Constant Runtime over $\mathbb{Z}$). *Constant runtime of linear loops over $\mathbb{Z}$ with eigenvalues from $\{-1, 0, 1\}$ is decidable.*

5 Related Work

Concerning related work on complete techniques, decidability of termination for linear single-path loops over $\mathbb{R}$, $\mathbb{Q}$, and $\mathbb{Z}$ was proven in [28], [7], and [17], respectively. Furthermore, [30] and [15] presented decidability results on termination of solvable loops and of triangular weakly non-linear (twn) loops, respectively. In contrast to the loops considered in our paper, such loops allow restricted forms of non-linearity. Building on [15], a technique to solve the (non-universal) halting problem (i.e., to decide termination for a *fixed* input), and to infer polynomial runtime bounds for twn-loops was introduced in [14]. The complete techniques for termination analysis and inference of upper bounds for twn-loops from [14, 15] were implemented in the tool KoAT [20, 21].

Our work complements these results by providing a decision procedure for constant runtime. Similar to [21, Lemma 22], our approach can be extended to eigenvalues which are roots of reals (or of $\{-1, 0, 1\}$, in the integer case), i.e., to eigenvalues λ where $\lambda^n \in \mathbb{R}$ (or $\lambda^n \in \{-1, 0, 1\}$) for some $n \in \mathbb{N}$. In the future, it would be interesting to lift our results to update matrices with arbitrary eigenvalues (including complex ones that are no roots of reals). However, in such a setting, several steps of our technique must be revised. For example, the number

of roots of poly-exponential expressions involving complex numbers cannot be bounded by a constant as in Sect. 3.3 and the quantifier elimination of Sect. 3.4 to allow the subsequent application of Fourier-Motzkin elimination would no longer work. Similarly, it would be interesting to generalize our results for $\mathbb{Z}$.

Note that the techniques mentioned above rely on the fact that the asymptotically dominant addend of closed-form expressions like (PE) can easily be determined. In Sect. 3.5, we use a similar reasoning, but there, we are *not* interested in the asymptotic behavior of (PE), but whether an inequation eventually gives rise to a lower or upper bound on a given variable x . Thus, we are just interested in the dominant addend of x 's *coefficient* when regarding (PE) as a linear polynomial with poly-exponential coefficients. Hence, our approach differs fundamentally from earlier techniques for deciding termination. In particular and in contrast to earlier approaches (and as already mentioned in Sect. 1), we cannot exploit the “eventual monotonicity” of linear loops with real eigenvalues, as the initial number of “non-monotonic” steps (when ignoring the loop guard) is often *not* bounded by a constant, even if the runtime of the loop is constant. To see this, consider the following loop with constant runtime:

while $0 \leq x \leq 10$ **do** $\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ **end**

Here, x behaves monotonically as soon as y becomes non-negative (as $y \geq 0$ implies $x \leq \text{up}(x)$). However, the number k such that $\text{up}^k(y) \geq 0$ holds is unbounded, as it depends on the initial value of y .

The complete technique for computing polynomial runtime bounds for twn-loops from [14] allows for computing a polynomial upper bound for any terminating twn-loop. However, these bounds are not necessarily tight, i.e., the resulting bound might be non-constant even if the runtime of the loop is constant. Thus, our work is orthogonal to [14].

There exist several techniques and tools for automatic complexity analysis of programs (with arbitrary multiple loops, if-then-else, non-determinism, ...) where the variables range over $\mathbb{Z}$. Most of these tools, e.g., CoFloCo [9], KoAT [20], Loopus [27], MaxCore [2], and RAML [16], infer *upper* bounds on the worst-case runtime complexity. Thus, if such a tool computes a constant runtime bound, then the program is guaranteed to terminate within a constant number of steps. In contrast, there also exist tools like LOBER [3] or LoAT [12], which infer *lower* bounds on the worst-case runtime complexity of loops or programs, respectively. Consequently, such tools can *disprove* constant runtime.

6 Evaluation and Conclusion

Evaluation: To evaluate the practical applicability of our decision procedure, we implemented the approach from Sect. 3 in Python, using SymPy [24] for all computations with algebraic numbers. We did not yet implement the adaption to loops over $\mathbb{Z}$ from Sect. 4, as we are not aware of any implementation of a decision procedure for one-parametric Presburger arithmetic. For our evaluation, we used the benchmark set from [11] consisting of all 1495 linear single-path loops

from the category *Termination of Integer Transition Systems* of the *Termination Problems Data Base* [29], the benchmark suite used in the annual *Termination and Complexity Competition (TermComp)* [13]. After removing duplicates, we obtained 1336 loops. All these loops have only real eigenvalues, which indicates that our approach is indeed applicable to a large class of examples. 1327 of them have eigenvalues from $\{-1, 0, 1\}$, which shows that even our seemingly restrictive result for loops over $\mathbb{Z}$ (see Sect. 4) is very relevant from a practical point of view.

For our experiments, we compared the performance of our tool **Loopy** against all tools for complexity analysis of integer transition systems that participated in this year's *TermComp*, i.e., **CoFloCo**, **KoAT**, and **LoAT**. Note that these three tools consider initial values from $\mathbb{Z}$ instead of $\mathbb{R}$ or $\mathbb{Q}$. Thus, if **LoAT** infers a non-constant *lower* bound over the integers for a loop, then this loop also has non-constant runtime over the rationals and reals; conversely, whenever **Loopy** proves a constant bound over the rationals and reals, then this is also a constant *upper* bound over the integers. To the best of our knowledge, there are no other tools which analyze complexity of programs over $\mathbb{R}$ or $\mathbb{Q}$. **Loopy** proved constant runtime for 76 loops and non-constant runtime for the 1260 remaining loops, with an average analysis time of 1.71 seconds on an AMD Ryzen 7 3700X octa-core CPU. This shows that our decision procedure is not only of interest from a theoretical point of view, but it is also efficient on a standard benchmark set, which indicates its applicability in practice. **CoFloCo** and **KoAT** were able to prove constant runtime for five and six of these 76 loops, respectively, whereas **LoAT** proved non-constant runtime for 1226 loops over $\mathbb{Z}$. All examples that were classified as non-constant by **LoAT** were also classified as non-constant by **Loopy**, i.e., there are no conflicting results. Note that **CoFloCo** and **KoAT** do not implement dedicated techniques for proving constant bounds, and thus it is not surprising that they do not infer more constant runtime bounds. Hence, our experiments demonstrate that tools for upper bounds can benefit from our decision procedure to improve their precision on such loops. To download **Loopy** and for the detailed results of our evaluation, we refer to [22].

Conclusion: We presented the first complete approach for the constant runtime problem of linear loops. For loops over $\mathbb{R}$ and $\mathbb{Q}$, our decision procedure applies if the eigenvalues of the update matrix are real. For loops over $\mathbb{Z}$, it covers eigenvalues from $\{-1, 0, 1\}$. Our empirical evaluation shows the practicability of our approach. In future work, we will try to add support for initial conditions for loops with real eigenvalues, which would yield a partial solution to the long-standing open problem whether the existence of multiphase-linear ranking functions is decidable for linear loops [5]. Note that while termination of linear loops for fixed initial values corresponds to the positivity problem (which has been open for decades), the positivity problem is decidable for real eigenvalues [1]. Moreover, we will try to extend our approach to arbitrary complex eigenvalues (without considering initial conditions).

Data Availability Statement

An artifact including our implementation in the tool *Loopy* is available at [22]:

<https://doi.org/10.5281/zenodo.17313899>

This artifact allows to reproduce all results of our evaluation.

References

1. Akshay, S., Balaji, N., Vyas, N.: Complexity of restricted variants of Skolem and related problems. In: Proc. MFCS '17. LIPIcs 83 (2017). <https://doi.org/10.4230/LIPICS.MFCS.2017.78>
2. Albert, E., Bofill, M., Borralleras, C., Martín-Martín, E., Rubio, A.: Resource analysis driven by (conditional) termination proofs. Theory and Practice of Logic Programming **19**(5-6), 722–739 (2019). <https://doi.org/10.1017/S1471068419000152>
3. Albert, E., Genaim, S., Martín-Martín, E., Merayo, A., Rubio, A.: Lower-bound synthesis using loop specialization and Max-SMT. In: Proc. CAV '21. pp. 863–886. LNCS 12760 (2021). https://doi.org/10.1007/978-3-030-81688-9_40
4. Ben-Amram, A.M., Genaim, S.: On multiphase-linear ranking functions. In: Proc. CAV '17. pp. 601–620. LNCS 10427 (2017). https://doi.org/10.1007/978-3-319-63390-9_32
5. Ben-Amram, A.M., Doménech, J.J., Genaim, S.: Multiphase-linear ranking functions and their relation to recurrent sets. In: Proc. SAS '19. pp. 459–480. LNCS 11822 (2019). https://doi.org/10.1007/978-3-030-32304-2_22
6. Bogart, T., Goodrick, J., Woods, K.: Parametric Presburger arithmetic: logic, combinatorics, and quasi-polynomial behavior. Discrete Analysis (2017). <https://doi.org/10.19086/da.1254>
7. Braverman, M.: Termination of integer linear programs. In: Proc. CAV '06. pp. 372–385. LNCS 4144 (2006). https://doi.org/10.1007/11817963_34
8. Cai, J.Y.: Computing Jordan normal forms exactly for commuting matrices in polynomial time. International Journal of Foundations of Computer Science **5**(3/4), 293–302 (1994). <https://doi.org/10.1142/S0129054194000165>
9. Flores-Montoya, A.: Upper and lower amortized cost bounds of programs expressed as cost relations. In: Proc. FM '16. pp. 254–273. LNCS 9995 (2016). https://doi.org/10.1007/978-3-319-48989-6_16
10. Frohn, F., Giesl, J.: Termination of triangular integer loops is decidable. In: Proc. CAV '19. pp. 426–444. LNCS 11562 (2019). https://doi.org/10.1007/978-3-030-25543-5_24
11. Frohn, F.: A calculus for modular loop acceleration. In: Proc. TACAS '20. pp. 58–76. LNCS 12078 (2020). https://doi.org/10.1007/978-3-030-45190-5_4
12. Frohn, F., Giesl, J.: Proving non-termination and lower runtime bounds with LoAT (System description). In: Proc. IJCAR '22. pp. 712–722. LNCS 13385 (2022). https://doi.org/10.1007/978-3-031-10769-6_41
13. Giesl, J., Rubio, A., Sternagel, C., Waldmann, J., Yamada, A.: The termination and complexity competition. In: Proc. TACAS '19. pp. 156–166. LNCS 11429 (2019). https://doi.org/10.1007/978-3-030-17502-3_10, website of *Term-Comp*: https://termination-portal.org/wiki/Termination_Competition
14. Hark, M., Frohn, F., Giesl, J.: Polynomial loops: Beyond termination. In: Proc. LPAR '20. pp. 279–297. EPiC 73 (2020). <https://doi.org/10.29007/NXV1>

15. Hark, M., Frohn, F., Giesl, J.: Termination of triangular polynomial loops. *Formal Methods in System Design* **65**(1), 70–132 (2025). <https://doi.org/10.1007/s10703-023-00440-z>
16. Hoffmann, J., Das, A., Weng, S.: Towards automatic resource bound analysis for OCaml. In: *Proc. POPL '17*. pp. 359–373 (2017). <https://doi.org/10.1145/3009837.3009842>
17. Hosseini, M., Ouaknine, J., Worrell, J.: Termination of linear loops over the integers. In: *Proc. ICALP '19. LIPIcs* 132 (2019). <https://doi.org/10.4230/LIPIcs.ICALP.2019.118>
18. Kincaid, Z., Breck, J., Cyphert, J., Reps, T.: Closed forms for numerical loops. *Proceedings of the ACM on Programming Languages* **3**(POPL) (2019). <https://doi.org/10.1145/3290368>
19. Kovács, L.: A complete invariant generation approach for P-solvable loops. In: *Proc. PSI '09*. pp. 242–256. LNCS 5947 (2010). https://doi.org/10.1007/978-3-642-11486-1_21
20. Lommen, N., Meyer, F., Giesl, J.: Automatic complexity analysis of integer programs via triangular weakly non-linear loops. In: *Proc. IJCAR '22*. pp. 734–754. LNCS 13385 (2022). https://doi.org/10.1007/978-3-031-10769-6_43
21. Lommen, N., Giesl, J.: Targeting completeness: Using closed forms for size bounds of integer programs. In: *Proc. FroCoS '23*. pp. 3–22. LNCS 14279 (2023). https://doi.org/10.1007/978-3-031-43369-6_1
22. *Loopy: Artifact, Evaluation, and Benchmarks*. Zenodo (2025). <https://doi.org/10.5281/zenodo.17313899>
23. Mansutti, A., Starchak, M.R.: One-parametric Presburger arithmetic has quantifier elimination. In: *Proc. MFCS '25. LIPIcs* 345 (2025). <https://doi.org/10.4230/LIPICS.MFCS.2025.72>
24. Meurer, A., Smith, C.P., Paprocki, M., Čertík, O., Kirpichev, S.B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J.K., Singh, S., Rathnayake, T., Vig, S., Granger, B.E., Muller, R.P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M.J., Terrel, A.R., Roučka, Š., Saboo, A., Fernando, I., Kulal, S., Cimrman, R., Scopatz, A.: *SymPy: symbolic computing in Python*. *PeerJ Computer Science* **3**, e103 (2017). <https://doi.org/10.7717/peerj-cs.103>
25. Ouaknine, J., Pinto, J.S., Worrell, J.: On termination of integer linear loops. In: *Proc. SODA '15*. pp. 957–969 (2015). <https://doi.org/10.1137/1.9781611973730.65>
26. Pólya, G., Szegő, G.: *Problems and Theorems in Analysis II: Theory of Functions. Zeros. Polynomials. Determinants. Number Theory. Geometry. Classics in Mathematics*, Springer (1976). <https://doi.org/10.1007/978-3-642-61905-2>
27. Sinn, M., Zuleger, F., Veith, H.: Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of Automated Reasoning* **59**(1), 3–45 (2017). <https://doi.org/10.1007/s10817-016-9402-4>
28. Tiwari, A.: Termination of linear programs. In: *Proc. CAV '04*. pp. 70–82. LNCS 3114 (2004). https://doi.org/10.1007/978-3-540-27813-9_6
29. TPDB (Termination Problems Data Base), <https://github.com/TermCOMP/TPDB>
30. Xu, M., Li, Z.B.: Symbolic termination analysis of solvable loops. *Journal of Symbolic Computation* **50**, 28–49 (2013). <https://doi.org/10.1016/j.jsc.2012.05.005>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Integrating String Reasoning in Symbolic Execution of C Programs

Rachel Cleaveland^{ID} and Clark Barrett^{ID}

Stanford University, Stanford, CA, USA
{rcleavel,barrettc}@stanford.edu

Abstract. Symbolic execution is a powerful program analysis technique which explores new paths through a program by generating inputs that are guaranteed to exercise those paths. However, there is a lack of modern research on symbolic execution for programs that manipulate strings, especially for C programs. In this work, we develop an approach for symbolically executing string-manipulating C programs and implement it in a proof-of-concept tool called SYMCC-STR. SYMCC-STR is built on top of the concolic execution engine SYMCC and the `smt-switch` SMT solving API, enabling it to make use of modern advancements in both symbolic execution and string solving. SYMCC-STR’s novelty lies in its *hybrid* data representation, expressing symbolic data *both* as SMT bitvectors *and* strings in order to balance the performance and expressivity of these respective theories. Through this novel approach, SYMCC-STR achieves a median 9.9% increase in branch coverage (up to 990% in the best case) compared to SYMCC on 6 real-world benchmark programs.

Keywords: Symbolic execution · Theory of Strings · SMT.

1 Introduction

Symbolic execution is a program analysis technique which enables the simultaneous exploration of multiple program paths by representing user-defined inputs symbolically. Constraints are added to these symbolic inputs to represent program execution and control flow, and these constraints can be solved to produce new inputs that are guaranteed to explore particular paths through the program.

The efficacy of a symbolic execution engine is contingent upon its ability to accurately convert a program’s entire semantics into constraints. As such, any symbolic execution engine aiming to completely encode the space of program behaviors must be able to accurately represent strings, which are a fundamental datatype in modern programming languages. Strings are often used to represent user-provided program data [4]. After being read into the program, these inputs may be subject to string-specific manipulations and comparisons, including regular expression membership checks, comparison of individual characters or substrings, and deletion or replacement of substrings.

Vulnerabilities enabled by the use of such string manipulations can have major consequences for the security of the programs that employ them. Notable

string-enabled vulnerabilities include XSS attacks [23] and SQL injection attacks [25], both of which arise from malicious string inputs to a program that allow untrusted users to execute code or perform database queries, respectively. Symbolic execution engines that are overly conservative when reasoning about strings can miss finding such critical vulnerabilities during their analyses.

Although reasoning about strings in symbolic execution is not a new topic [5, 7, 21, 33, 39, 43, 45, 48, 49], previous work suffers from several limitations.

First, the symbolic execution engines in previous work were mainly designed for the analysis of Java programs, meaning they are not applicable to programs in other languages such as C. The importance of analyzing C code in particular is evidenced by large-scale projects like Google’s OSS-Fuzz [47], which performs continuous fuzzing on open-source C/C++ projects.

Second, much of this work is now out-of-date, in that it was done when specialized solvers for string constraints were either nonexistent or much more primitive than they are now [27]. In fact, a main focus of prior work is the creation (often from scratch) of basic string solving capabilities needed for handling the symbolic execution task [8, 21, 39, 45, 49]. However, since then, there have been vast improvements in general-purpose string solvers [15–17, 29–32, 40–43]. In particular, many SMT solvers now include efficient and powerful string theory solvers. These advancements have been crucial to large-scale deployments of string solving in industrial applications [6, 44].

Third, prior tools have mostly been evaluated on small test programs limited to a few hundred lines of code [5, 33, 39, 49]. Such evaluations provide little insight on the applicability of these tools to real-world programs, which may contain tens to hundreds of thousands of lines of code. Thus, the question of scalability has largely been left unanswered.

Motivated by these limitations, this paper revisits the topic of symbolic execution for string-manipulating programs in order to answer the following questions: What are the modern bottlenecks to a string-based symbolic execution approach, and what are ways to address these? We develop our approach into a proof-of-concept symbolic execution engine, which we call SYMCC-STR, targeting string-manipulating C programs. SYMCC-STR uses `smt-switch` [35], a generic SMT solving API that allows it to take advantage of modern advancements in string solving. Furthermore, our design of SYMCC-STR is motivated by the following two performance-related insights that enable SYMCC-STR to scale to larger benchmarks.

Our first insight is that programs often perform operations that do not require the full expressivity of the theory of strings to encode. Thus, in SYMCC-STR we implement a novel “hybrid” data representation, whereby input data is represented as *both bitvectors and strings*. SYMCC-STR always first attempts to solve problems using the bitvector representation, and only if that fails to produce a new program input does it attempt to solve the problem using the string representation. This enables significantly better performance than a string-only representation would, while generating more new inputs than would be possible using only a bitvector representation.

Our second insight is that back-end string solvers, despite benefiting from vast improvements in the past decade, are still a limiting factor during execution. To address this, we first develop and implement novel string assertion rewrites and a custom string inequality solving pass within SYMCC-STR to bolster the solver. However, even with these contributions, the assertions often eventually become too large and complex for the string solver to solve. Therefore, we enhance this with a mechanism that simply turns off the string solver when it stops generating new inputs.

These innovations enable SYMCC-STR to analyze large benchmarks—with sizes ranging between 550 and 68,000 lines of code—while improving branch coverage by a median of 9.9% (up to 990% in the best case) over SYMCC. We summarize our contributions as follows:

- We develop SYMCC-STR, the first symbolic execution engine for string-manipulating C programs that uses a novel hybrid data representation, representing symbolic data as both bitvectors and strings.
- We perform a case study on 6 real-world programs from the OSS-Fuzz project suite, demonstrating SYMCC-STR’s performance on large benchmarks.
- We identify new string assertion rewrites and implement a novel string inequality-solving pass within SYMCC-STR in order to performantly handle constraints containing string comparisons.
- We package the solver queries generated from our case study into a challenging new set of string benchmarks for the SMT community.

2 Background

2.1 Symbolic Execution

Symbolic execution is a technique for representing and reasoning about multiple possible program executions by: (i) encoding inputs as symbolic variables; (ii) generating constraints over these variables to represent the execution of the program; and (iii) solving these constraints to produce real inputs to the program that follow the execution path encoded by the constraints.

Symbolic execution engines come in many varieties, differing, for example, in their execution model (are programs-under-test interpreted [13, 18, 50] or dynamically run [14, 22, 37, 38]), their memory model (are symbolic addresses concretized [12]), or the types of programs they accept (source code [13, 38] or binaries [14, 18, 22, 50]). One thing that all engines have in common, however, is that they suffer from the *path explosion problem*, which limits their scalability. A number of approaches have been explored to address this challenge.

One such technique is concolic execution, in which a concrete input is dynamically run through a program, and a symbolic query is solved for each branch encountered to try to generate an input that takes the opposite branch from the one taken during the execution. In this way, concolic execution does not explore the entire program at once, and new inputs can be fed back into the engine to

```

1  mov    %r14,%rdi
2  mov    0x20(%rsp),%rsi
3  call   401180 <_sym_build_add@plt> # Symbolically add one to x
4  mov    %rax,-0xc8(%rbp)
5  add    %r12,1 # Concretely add one to x
6  mov    %r12,-0xcc(%rbp)
7  mov    $0x2,%rdi
8  call   401130 <_sym_build_integer@plt>
9  mov    -0xc8(%rbp),%rdi
10 mov    %rax,%rsi
11 call   4010f0 <_sym_build_signed_greater@plt> # Make branch constraint
12 mov    %rax,-0xe8(%rbp)
13 mov    -0xcc(%rbp),%eax
14 cmp    $0x2,%eax # Evaluate concrete branch condition
15 setg   %al
16 mov    %al,-0xd9(%rbp)
17 mov    -0xe8(%rbp),%rdi
18 mov    %al,%esi
19 mov    $0x1bfe9f0,%edx
20 call   4010b0 <_sym_push_path_constraint@plt> # Generate new input
21 mov    -0xd9(%rbp),%al
22 test   $0x1,%al
23 jne    40171a <main+0x48a> # Take concrete branch

```

Fig. 1: (Simplified) Assembly code after SYMCC instrumentation of one line of C code (`if (x + 1 > 2) {...}`). Black instructions are original program instructions, and blue instructions are those inserted by SYMCC. Register `%r14` initially holds the symbolic expression for `x`, `%r12` the concrete expression for `x`, and `0x20(%rsp)` a symbolic expression representing the constant 1.

iteratively explore more paths. This technique requires fewer resources than the purely symbolic approach and easily couples with fuzz testing [51].

Another optimization, introduced in the concolic engine SYMCC [38], is to compile the symbolic execution functionality directly into the program-under-test. SYMCC is implemented as an LLVM pass and runtime library. It instruments the LLVM IR of a program-under-test to directly insert calls to functions within its runtime library that construct and solve symbolic path constraints.

To understand this instrumentation process, consider Fig. 1, which shows the original assembly for the line of code `if (x + 1 > 2)` in black and the assembly added by SYMCC in blue. Each time a concrete computation is performed (e.g., the addition of 1 to register `%r12`, which contains `x`, on line 5), a corresponding symbolic operation is done (e.g., the call to function `_sym_build_add` on line 3). Each time a concrete conditional branch is executed (e.g., line 23), the corresponding symbolic condition is generated (e.g., line 11). Its negation is then sent to the solver, and the corresponding query is solved to obtain a new input that takes the opposite path from the concrete input, if one exists (e.g., line 20). SYMCC exhibits superior performance compared to engines that interpret, rather than natively execute, their programs-under-test.

Data Representations Developers of symbolic execution engines make different decisions about how to represent symbolic data and memory. Most engines represent symbolic data either as bitvectors (e.g., [38, 50]) or arrays of bitvectors (e.g., [13, 19]). Both representations have drawbacks when it comes to encoding strings. Bitvectors are easily mutable, but must be of a fixed length. Strings

in C are not required to be any particular length, so it is suboptimal to encode them using fixed-length variables. Arrays do not have a fixed length; however, repeated array updates result in prohibitively large assertions. Both data representations lack native support for important string-specific functions like `indexOf` and `strcmp` (i.e., lexicographic comparison) [11].

2.2 String Solvers

String solvers are used to determine the satisfiability of constraints involving string variables and constants [3]. If a string solver returns `sat`, then the set of constraints has a solution, i.e., an assignment of concrete strings to variables that makes each constraint evaluate to true. On the other hand, an `unsat` result means that no such assignment exists.

Early developments in string solving were largely motivated by the need for test-case generation and verification in the realm of web security [3, 26, 36, 39, 45, 48]. However, these initial solvers suffered from various limitations. Many supported only strings with fixed lengths [28, 45, 46], and many did not support common operations (e.g. `contains`, `indexOf`). Moreover, solvers capable of reasoning about unbounded-length strings often relied on reductions to automata, which historically limited their performance or expressiveness [4]. These weaknesses limited the ability of such early solvers to reason about real string-manipulating programs.

However, in the past decade, string solvers have improved significantly, both in their performance and in their expressivity [15, 17, 20, 30, 52]. Many of these improvements were motivated by benchmarks generated from symbolic execution applications [29, 40, 41, 43] (e.g., the addition of extended string functions like `indexOf`, as well as the support for string-to-code point conversion [40, 41, 43]). Despite the fact that support for a full set of program-level string operations leads to undecidability [11], modern string solvers are often capable of solving constraints generated by complex real-world programs. This is evidenced by applications like Zelkova, which calls string solvers billions of times a day to secure cloud data at Amazon Web Services [6, 34, 44].

3 Related Work

As mentioned, string-based attacks like SQL injection [25] and XSS [23] motivated earlier work on symbolic execution of string-manipulating Java/JavaScript programs [8, 27, 36, 39, 48, 49]. However, our work is (to the best of our knowledge) the first to address string-manipulating C programs. Reasoning about strings in C presents unique challenges. For one, strings in C are not a built-in type, instead being represented as pointers to null terminated memory regions. Accounting for this null terminator complicates the representation of symbolic memory. Another complication is that C is not a strongly-typed language, meaning the same data can be interpreted as multiple types (e.g., data can be read into the program

```

1 char buffer[10000];
2 memset(buffer,0,10000);
3 read(STDIN_FILENO, buffer, sizeof(buffer) - 1);
4 if (strlen(buffer) == 9999) { /* Code A */ }
5 if (buffer[3] == 'a') { /* Code B */ }

```

Listing 1.1: Code snippet showing branches that SYMCC-STR can cover, given an empty initial input to `stdin`, but SYMCC cannot.

as a string, and then converted to an integer via the `atoi` function). Addressing this required minimizing type conversions, which we discuss in §5.1.

Another difference is that the engines in [8, 21, 39, 45, 48, 49] all implement custom string solvers in their back ends. In other words, the string solver itself (or an extension to a custom string solver from prior work) is a primary contribution. In contrast, SYMCC-STR uses `smt-switch` [35] as its back-end solver API, allowing it to easily connect to any SMT solver that supports the SMT-LIB [10] standard. Thus, where prior work focused on developing new string solvers, this work seeks to scale symbolic execution of string programs to real-world software, uncovering the limitations of modern string solvers in the process.

Finally, on the topic of scalability, most prior work was limited to handling benchmarks of only tens or hundreds of lines of code [5, 21, 33, 39, 45, 49]. In contrast, our benchmarks range from 550 to 68,000 lines of code, representing a major advancement in scalability.

4 SYMCC-STR Overview

In this section, we introduce SYMCC-STR, a concolic execution engine for string-manipulating C programs designed to answer two main research questions: what bottlenecks still exist for symbolic execution using modern string solvers, and how can these be addressed? SYMCC-STR is derived from SYMCC [38], with the key change that symbolic data is represented *both* as strings *and* bitvectors. We explain the rationale for this below.

4.1 Drawbacks of a Bitvector-Only Approach

The theory of fixed-size bitvectors includes many bitwise logical and arithmetic operations, like `bvand` and `bvmul`, making this theory a natural choice to represent low-level data during concolic execution. However, there are drawbacks to the use of bitvectors when it comes to symbolically representing strings.

Symbolic Input Lengths SYMCC replaces the concrete inputs during execution with symbolic bitvector variables, whose length must be provided upon their creation. SYMCC simply assigns these variables the length of their corresponding concrete inputs. As a result, it is unable to generate test cases with inputs that are different in length from the original concrete input.

For example, consider the code in Listing 1.1, and suppose the concrete input provided is the empty string. SYMCC cannot generate a test case that explores

Code B on line 5 because it only considers symbolic inputs of length 0. It is possible to get around this limitation by manually setting the size of a concrete input, but manually exploring the large space of potential sizes is expensive, and in general, it is impossible to explore all of it. In contrast, string variables can have any length—the size does not have to be chosen up front. By using a string representation, new inputs can be generated whose lengths are different from the concrete input (e.g., the input `AAAA` could be generated to explore **Code B**).

Hooked Functions To efficiently execute certain `libc` functions symbolically, SYMCC replaces calls to these functions during compilation (e.g., `malloc`) with custom runtime functions, or *hooked functions*, which efficiently encode the execution of the original `libc` functions. However, SYMCC cannot do this for `libc` string functions.

Consider again Listing 1.1. SYMCC does not hook the `strlen` on line 4 because the bitvector encoding of a string always has a fixed length. Thus, SYMCC has no useful symbolic expression to assign as its symbolic result. The execution can proceed in one of two ways. If `libc` is not compiled with SYMCC, then `strlen` will be executed fully concretely, and its return value will always match the concrete execution value (i.e., it will not be symbolic). Alternatively, if `libc` is compiled with SYMCC, then the source code of `strlen` will be symbolically executed. The code returns the value of a counter that is incremented in a loop until the input’s null terminator is reached. Within the loop, SYMCC can generate test cases with different placements of the null terminator that thus produce different values for `strlen`. However, this choice is guided by the branches of the loop inside the `strlen` function, not by the branch condition on line 4. Thus, it could take many runs of SYMCC to eventually generate an input correctly formatted to explore **Code A**.

With a string representation of symbolic data, however, calls to `strlen` can be hooked by simply applying the string length operator to the symbolic input (i.e., `len(buffer)`). Solving the query generated by the branch condition on line 4 could then produce a new input exploring **Code A** in a single run.

4.2 Hybrid Data Representation

Our preliminary testing of SYMCC-STR revealed that the string-only representation of data also has significant drawbacks, mostly in terms of prohibitively slow performance (despite modern string solving advancements—see §6). We thus propose a “hybrid” approach, wherein *both* bitvector *and* string representations of data are tracked for each symbolic variable.

SYMCC-STR maintains two separate sets of symbolic expressions and assertions during execution. Wherever SYMCC would create a symbolic expression, SYMCC-STR creates two expressions: one based on bitvectors and one based on strings. Similarly, wherever SYMCC would assert a formula, SYMCC-STR asserts two formulas, one using bitvectors and the other using strings, within two separate SMT solver instances.

The advantages of our hybrid approach are two-fold. First, in practice, many queries do not require the additional expressivity of the theory of strings to be

	sat	unsat	unknown	timeout	Time (s)
cvc5	112	95	0	93	1220
z3	65	92	0	143	1540
ostrich	35	81	1	183	3418
z3-noodler	29	90	0	181	1683

Table 1: String solver comparison on the first 300 benchmarks of our benchmark set [1] (see §7), 50 per case study.

solved. For example, the condition `buffer[3] == 'a'` on line 5 of Listing 1.1 can easily be satisfied *if* the initial input is at least four bytes long. Therefore, in our hybrid approach, we always let the bitvector solver make the first solving attempt. Only if the bitvector solver cannot generate a new input (i.e., returns **unsat**) do we invoke the string solver. This enables the best of both worlds: the performance of the bitvector solver in many cases, and the additional expressivity of the string solver when necessary.

The second advantage of the hybrid approach is that it enables SYMCC-STR to selectively disable the string solver. We observed that the string solver is very useful early on in larger programs, but it eventually becomes unable to solve the assertions in a reasonable amount of time. SYMCC-STR thus disables the string solver once its utility diminishes beyond a certain point (see §6.3 for details).

4.3 String Solver Extensions

One of the goals of this work was to assess the capabilities of modern string solvers and find opportunities for their improvement. By analyzing the SMT queries generated by SYMCC-STR, we identified several specific opportunities for solver improvement, and implemented workarounds for them in SYMCC-STR directly. In our work and evaluation (see §6), we used cvc5 [9] as a back-end solver, as it showed the most promising performance in preliminary testing. This was confirmed after running cvc5, z3 [20], ostrich [16], and z3-noodler [17] on a subset of our benchmark set [1] (see §7), showing cvc5 to be the best performer for our application (Tab. 1). We hope the developers of cvc5 will incorporate these optimizations in future versions of the solver. We expect them to be useful for other string solvers as well, though we leave that evaluation to future work.

Our first optimization focuses on patterns of assertions like the following:

$$\text{substr}(S, 0, 9) < \text{"Document"} \quad \wedge \quad \text{"Feature"} < \text{substr}(S, 0, 8)$$

Because it is always true that $\text{substr}(S, 0, 8) \leq \text{substr}(S, 0, 9)$, these assertions create a cycle of inequalities, two of which are strict, and are thus **unsat**. However, cvc5 times out on these assertions (with the 10-second timeout we used in SYMCC-STR). Further investigation revealed two limitations of cvc5’s string solver. First, cvc5 cannot quickly deduce the relationship between “Document” and “Feature”, meaning that even when the substrings in the two assertions are exactly the same, cvc5 still cannot solve the problem in a reasonable amount of time. Second, cvc5 cannot deduce that for any trio of strings S_1 , S_2 , and S_3 such that $S_1 = S_2 ++ S_3$, $S_2 \leq S_1$. That is, a string is always greater than or equal (lexicographically) to any of its prefixes.

$\text{indexOf}(\text{con}(A, B), c, 0) \rightarrow -1$	$\text{if } \text{len}(c) = 1$
	$\wedge \text{indexOf}(A, c, 0) = -1$
	$\wedge \text{indexOf}(B, c, 0) = -1$
$\text{indexOf}(\text{con}(A, B), c, 0) \rightarrow \text{indexOf}(A, c, 0)$	$\text{if } \text{indexOf}(A, c, 0) \neq -1$
$\text{indexOf}(\text{con}(A, B), c, 0) \rightarrow \text{len}(A) + \text{indexOf}(B, c, 0)$	$\text{if } \text{len}(c) = 1$
	$\wedge \text{indexOf}(A, c, 0) = -1$
	$\wedge \text{indexOf}(B, c, 0) \neq -1$
$\text{substr}(A, x, y) \rightarrow A$	$\text{if } x = 0 \wedge \text{len}(A) = y$
$\text{substr}(\text{con}(A, B), x, y) \rightarrow \text{substr}(A, x, y)$	$\text{if } x + y \leq \text{len}(A)$
$\text{substr}(\text{con}(A, B), x, y) \rightarrow \text{substr}(B, x - \text{len}(A), y)$	$\text{if } x \geq \text{len}(A)$
$\text{substr}(\text{substr}(A, x, y), a, b) \rightarrow \text{substr}(A, x + a, b)$	$\text{if } a \geq 0 \wedge x \geq 0 \wedge a + b \leq y$
$\text{substr}(\text{substr}(A, x, y), a, b) \rightarrow \text{substr}(A, x + a, y - a)$	$\text{if } a \geq 0 \wedge x \geq 0 \wedge y \leq a + b$
$\text{len}(\text{substr}(A, x, y)) \rightarrow y$	$\text{if } \text{len}(A) \geq x + y \wedge y \geq 0 \wedge x \geq 0$
$\text{len}(\text{ITE}(\text{cond}, A, B)) \rightarrow \text{len}(A)$	$\text{if } \text{len}(A) = \text{len}(B)$
$A == B \rightarrow \text{false}$	$\text{if } \text{len}(B) = 1 \wedge \text{!contains}(A, B)$

Fig. 2: String-based simplifications implemented in SYMCC-STR.

Based on these observations, we implemented a novel string inequality refutation procedure. Our implementation is inspired by a similar procedure used in CVC4’s bitvector solver [24]. Each time a string comparison constraint is asserted, it is added to a graph, where each string expression is a node, and each less than (less than or equal to) inequality is represented as a strong (weak) edge between the corresponding nodes. We also add tautological edges between nodes as follows: we add edges between all string constants based on their concrete lexicographical ordering (e.g., “Document” < “Feature”), and we add edges between strings and their prefixes. We immediately return **unsat** (without calling the solver) if we detect a cycle with at least one strong edge. Otherwise, we call the back-end solver as usual.

Our second optimization is based on the observation that keeping assertions simple allows the string solver to be useful for longer. However, there are many simplifications that can be performed only if some additional information is deducible from the existing assertions. For example, $\text{len}(\text{ITE}(\text{cond}, A, B))$ can be rewritten to $\text{len}(A)$, for strings A and B , if $\text{len}(A) = \text{len}(B)$ is known to be true. To implement such *conditional rewrites*, SYMCC-STR queries the solver with the negation of the rewrite condition using a very small timeout (we call this a “mini-check”). If the solver returns **unsat**, then the condition holds, and the rewrite can be performed.

Fig. 2 shows the conditional rewrites we have implemented in SYMCC-STR. Each has been separately verified as a sound lemma using *cvc5*. We also implemented several arithmetic simplifications to reduce the number of ITEs needed when checking for overflow and implementing sign extension.

5 Implementation

SYMCC-STR is implemented on top of SYMCC. It consists of an LLVM compiler pass, written in 2,500 lines of code, and a runtime library, written in 10,000

```

1 #define isspace(c) __isctype((c), _ISspace)
2 #define __isctype(c, type) \
3   ((*__ctype_b_loc ())[(int) (c)] & (unsigned short int) type)
4
5 /* Strip whitespace chars off end of string. */
6 static char* ini_rstrip(char* s)
7 {
8   char* p = s + strlen(s);
9   while (p > s && isspace((__isctype)(*--p))) *p = '\0';
10  return s;
11 }
12
13 int ini_parse_stream(...) {
14   ...
15   start = ini_rstrip(start);
16   if (strchr(";", "#", *start)) { /* Line comment */ }
17   ...
18 }

```

Listing 1.2: Code snippet taken from the `inih` benchmark program, in which data is treated both as a string and as a bitvector.

lines of code. 1,500 lines of the compiler and 2,000 lines of the runtime come from SYMCC, and the rest implement our unique approach. The runtime is split into the bitvector and string runtimes, which treat symbolic expressions as bitvectors and strings, respectively. The new code in the runtime comes primarily from string-based implementations of the entire SYMCC symbolic execution framework (creating expressions, pushing to the solver, solving constraints, and generating output files, 3,000 LOC), the symbolic memory store (see §5.2, 1,300 LOC), and new hooks and helper functions for common string functions plus string-based implementations of existing function hooks (1,500 LOC). In this section, we discuss some noteworthy implementation choices within SYMCC-STR.

5.1 Mixed String-Integer-Bitvector Constraints

In C programs, the same data may be interpreted in different ways. Consider Listing 1.2, a (condensed) code snippet from `inih`, a small `.ini` file parser used in our evaluation (see §6). On line 15, the input string `start` is stripped of any whitespace characters as defined by `libc`'s `__isctype` function, and then `strchr` is called on the result. However, `__isctype` performs a bitwise-and operation on its input, which is not natively defined in the theory of strings. Thus, we require multiple symbolic representations of the input to encode all of its occurrences.

In general, expressions may mix bitvector, string, and integer operations. Converting back and forth between different representations hurts performance because solvers struggle with such conversions. Therefore, we limit the number of conversions we perform in two ways. First, we replace calls to these `__isctype`-based functions, which introduce unavoidable type conversions, with implementations that do not. In Listing 1.2, for example, we would replace `isspace` with an implementation that uses integer comparison of the input character. This can be symbolically represented using code-point conversion, natively defined in

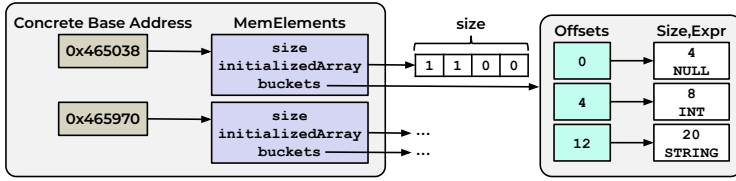


Fig. 3: SYMCC-STR’s symbolic memory store.

the theory of strings. This process should be easily automatable using front-end tools; we leave such automation to future work. Second, when building symbolic expressions themselves, we attempt to use operations from the same theory as the inputs whenever possible. For example, consider a scenario where bit-shifting is performed on an integer-typed symbolic expression. A naive approach would convert the integer to a bitvector, and then bit-shift the converted expression. However, in SYMCC-STR, we first check if the second argument is a constant, and if so, we perform the bit-shift using integer multiplication or division to avoid the type conversion. Future work could explore more ways to reduce the number of conversions. For example, taint tracking could be used to determine whether an input is used in more string or arithmetic expressions in the program and assign its type at runtime accordingly.

5.2 Symbolic Memory Store

SYMCC does not support symbolic addressing. If an array is accessed at a symbolic offset, SYMCC will use the concrete array entry accessed instead of considering all possible entries that could be accessed. In SYMCC-STR, however, we deemed symbolic addressing to be important in order to leverage the fact that symbolic strings do not have a fixed length. For example, performing a function like `strcat` on strings *A* and *B* could require placing string *B* at a symbolic memory address, as its location depends on the (possibly symbolic) length of *A*.

Our symbolic memory store maps base memory addresses to symbolic memory elements as is done in KLEE [13]. Our symbolic memory elements are designed to support symbolic offsetting. Specifically, when memory regions are represented using strings, reading the memory at a symbolic offset is as simple as returning the substring starting at the symbolic offset of the string in the memory store. Writing to the memory store involves replacing the portion of the stored string starting at the symbolic offset with the new string.

As mentioned above, we wish to avoid conversions when possible. Thus, we use a memory model that supports fine-grained heterogeneity. Each memory region is broken up into a map of concrete offsets, each of which can be of type `NULL` (i.e., concrete values only—no symbolic representation), `String`, `Int`, or `Bitvector`. However, if a read or write to a symbolic offset is done on a region that is split in this way, then we convert the entire region into a single string (by converting each subregion to a string and then concatenating all subregions) and then index into that string.

inih	550 LOC	INI parser	libspectre	4670 LOC	Postscript parser
libtasn1	30450 LOC	ASN.1 codec	libyaml	6760 LOC	YAML parser
openjpeg	68415 LOC	JPEG codec	speex	2817 LOC	Audio file codec

Table 2: Lines of code (LOC) and description of each benchmark.

5.3 Representing Uninitialized Memory

When SYMCC-STR encounters calls to C functions like `read`, `fread`, and `fgets`, it creates new symbolic variables to represent the input string. Typically, these strings are placed in a buffer that may be much larger than the value read. The rest of the buffer often remains uninitialized. This could be modeled by assigning a string variable to the full buffer and then writing over it. However, there are several drawbacks to this approach. For one, this introduces extra variables, which makes the problem harder to solve. Second, if the program ever does access uninitialized memory, the solver will try to use these variables to satisfy the path constraints, but since uninitialized memory is not actually controllable, this can produce false positives. This can be avoided by checking whether memory reads access uninitialized memory, but keeping track of which memory in a region is initialized versus uninitialized is difficult if the region is a concatenation of variables, some of which are implicitly uninitialized.

Due to these drawbacks, SYMCC-STR treats uninitialized memory as concrete, and does not assign it symbolic values. When an input string is read into an uninitialized buffer, this input string alone is placed into the memory store, regardless of whether it takes up the whole memory region. We thus avoid unnecessarily tracking extra symbolic variables and can easily check whether a memory read accesses uninitialized memory.

6 Evaluation

We evaluate SYMCC-STR on 6 programs from the OSS-Fuzz project suite: `inih`, `libspectre`, `libtasn1`, `libyaml`, `openjpeg`, and `speex`, summarized in Table 2. Each was selected for its use of common string library functions, like `strcmp` and `strlen`. Our evaluation is designed to answer the following research questions:

- **RQ1:** How does “hybrid” SYMCC-STR compare to running the bitvector back end only in terms of branch coverage, a customary evaluation metric for symbolic execution engines?
- **RQ2:** How do “pure-bv”, “pure-string”, and “hybrid” SYMCC-STR compare on a single run in terms of runtime and number of new test cases generated?
- **RQ3:** At what point is the string back end no longer useful (i.e., what is a reasonable trigger for disabling the string back end)?
- **RQ4:** How effective are the inequality solver and the conditional rewrites?

Recall that SYMCC-STR performs concolic execution, so it runs a single concrete test case and uses it to generate new test cases that diverge from the given test case at a single branch. In our experiments, we run SYMCC-STR either once

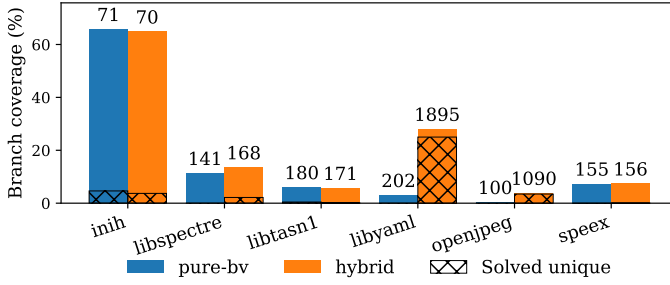


Fig. 4: Branch coverage of pure-bv and hybrid SYMCC-STR, repeatedly running each benchmark with the newly generated inputs for 12 hours. The absolute number of branches covered is given above each bar.

(i.e., a “single-run”), or in a loop which repeatedly feeds the newly generated test cases of each run back into the tool for 12 hours (i.e., a “multi-run”).¹

Note that we made the following minor changes to some benchmarks: we manually replaced re-implementations of common string functions with their `libc` counterparts (e.g., `dsc_strncmp` in `libspectre` with `strncmp`), and calls to the `__isctype` macros described in §5.1 with implementations that do not introduce bitwise operators. This was done in order to leverage our special hooked functions (§4.1) in the former case, and to minimize type conversions (§5.1) in the latter case so that SYMCC-STR can better demonstrate its potential. The automatic replacement of such functions and macros is a topic for future work.

6.1 RQ1: Branch Coverage of Pure-BV and Hybrid SYMCC-STR

In this experiment, we performed multi-runs of pure-bv SYMCC-STR and hybrid SYMCC-STR. We gave each single run a timeout of 5 minutes, and we disabled the string solver when 20% or fewer mini-checks succeeded (see §6.3).

The results are shown in Fig. 4. In most of the benchmarks, hybrid SYMCC-STR covers more branches than pure-bv SYMCC-STR, increasing the branch coverage by up to 990%, with a median increase of 9.9%. While four of the benchmarks show either a modest increase or a modest decrease in branch coverage, in the remaining two benchmarks, about 10x more branches are covered by hybrid than pure-bv SYMCC-STR. In these latter two cases, hybrid SYMCC-STR reaches a section of the code, after parsing, that the pure-bv engine was unable to reach. We conclude that hybrid SYMCC-STR, enabled by the string back end, was able to symbolically represent and solve enough branch conditions to successfully get through the parsing code, while pure-bv SYMCC-STR was not. This demonstrates that, at least in some cases, our hybrid approach can significantly improve on the pure-bv representation in terms of finding new paths to explore.

¹ We evaluate SYMCC-STR this way as opposed to coupling it with a fuzzer to isolate the work that SYMCC-STR is doing from any work that would be done by the fuzzer.

	pure-bv		pure-string		hybrid	
	sat	Runtime (s)	sat	Runtime (s)	sat	Runtime (s)
inih	701	7.3	106	17796	764	11364
libspectre	16911	17002	14	TO	386	TO
libtasn1	2741	102	140	TO	2763	TO
libyaml	2785	11105	56	TO	1636	TO
openjpeg	49	MO	63	TO	63	TO
speex	69	3.1	17	3962	80	3438

Fig. 5: Pure-bitvector, pure-string, and hybrid SYMCC-STR results on a single run with a representative input.

6.2 RQ2: Bitvector, String, and Hybrid Single-Run Comparisons

Next, we performed single runs of pure-bv, pure-string, and hybrid SYMCC-STR with a 12 hour timeout to better understand the tradeoffs of each version. The results are in Fig. 5. Pure-bv SYMCC-STR performs significantly better than pure-string SYMCC-STR due to its overall speed advantage. Pure-string SYMCC-STR times out on 4 benchmark programs, and on the other 2 benchmarks, it exhibits a 2,092x average increase in runtime. It also finds fewer **sat** instances, since it times out on many more queries than the pure-bv version. This is consistent with our observation that reasoning in the theory of strings is generally less efficient than in the theory of bitvectors.

The results for hybrid SYMCC-STR, however, are more encouraging. Across comparable benchmark results (i.e., those in which the pure-string and hybrid versions did not time out), hybrid SYMCC-STR finds 9.6% more **sat** instances on average than pure-bv SYMCC-STR and achieves a 32.0% average speedup in runtime compared to pure-string SYMCC-STR. These results are a promising first step toward trying to achieve the best of both approaches.

6.3 RQ3: Disabling SYMCC-STR’s String Side

Despite its improvement in runtime over pure-string SYMCC-STR, the runtime of hybrid SYMCC-STR in the experiment above is still impractical. We next report on a set of experiments designed to answer two questions: first, is the percentage of mini-checks being solved a useful heuristic for deciding when to turn off the string solver (described in §4.2), and if so, what is the performance impact of disabling the string solver at the optimal mini-check percentage?

For the first experiment, we recorded the percentage of mini-checks being solved along with the percentage of total **sat** instances found by the string and bitvector sides over time. These results (Fig. 7) show that for most benchmarks, all of the **sat** instances found by the string solver are found very early in the execution. Also, we typically see a sharp decline in the mini-check percentage solved similarly early, demonstrating a promising correlation between the mini-check solved percentage and the ability of the string solver to find new **sat** instances. Based on this analysis, we decided to use the 20% mark as a trigger for disabling the string solver.

	Baseline		20%	
	sat	Runtime (s)	sat	Runtime (s)
inih	764	11364	764	1089
libspectre	386	TO	16928	27876
libtasn1	2763	TO	2763	8296
libyaml	1636	TO	2802	16719
openjpeg	63	TO	57	MO
speex	80	3438	79	30

Fig. 6: Number of **sat** instances and runtime for hybrid SYMCC-STR, with the string side disabled once less than 20% of the mini-checks are being solved.

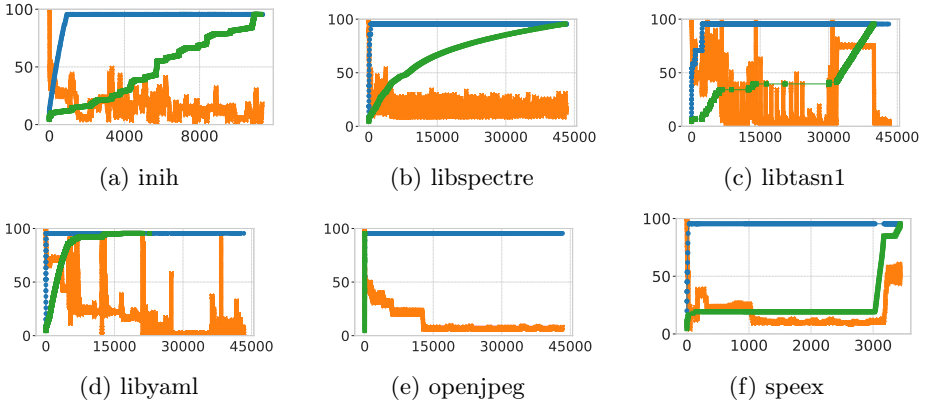


Fig. 7: Percentage of mini-checks being solved (orange), percentage of string-side **sat** instances found so far (blue), and percentage of bitvector-side **sat** instances found so far (green) over time.

The next experiment measures the impact of this decision. The results are shown in Fig. 6. In terms of **sat** instances found, there is almost no cost to disabling the string solver, and in some cases there is a large benefit. Moreover, there is a significant speedup: we observed up to a 99% reduction in runtime. Thus, this optimization is essential to keep the runtime of hybrid SYMCC-STR practical, enabling the branch coverage improvements we saw in §6.1.

6.4 RQ4: Inequality Solver and Conditional Rewrites

In the last set of experiments, we compare the effectiveness of the inequality solver and the conditional rewrites. The results for disabling the conditional rewrites are shown in Fig. 8. In most of the cases, enabling the rewrites results in a tradeoff: many more solutions at the cost of a longer runtime. This makes sense, as the rewrites add thousands of extra checks with a small timeout, adding up over the course of the execution. However, the rewrites also make the assertions easier to solve. Excluding **libspectre**, we observe a 63.5% increase in **sat** instances and a 9.8% increase in **unsat** instances.

libspectre is a clear outlier—with rewrites enabled, it finds significantly fewer **sat** and **unsat** instances, skewing the averages to a 53.7% increase and a

	No conditional rewrites			With conditional rewrites		
	sat	unsat	Runtime (s)	sat	unsat	Runtime (s)
inih	35	928	15972	106	937	17796
libspectre	24	2072	TO	14	1337	TO
libtasn1	68	2315	TO	140	2556	TO
libyaml	50	1924	TO	56	2172	TO
openjpeg	63	14	TO	62	14	TO
speex	17	73	3617	17	88	3962

Fig. 8: Ablation study for conditional rewrites.

	No inequality solver		With inequality solver	
	unsat	Runtime (s)	unsat	Runtime (s)
libspectre	246	3008	247	3007
libtasn1	2508	TO	2566	TO

Fig. 9: Ablation study for the inequality solver.

3.0% decrease in **sat** and **unsat** instances, respectively. An investigation revealed that this is partially due to a particular internal *cvc5* rewrite that gets applied when our conditional rewrites are enabled, but otherwise does not. The rewrite in question significantly degrades the performance of *cvc5* on this benchmark. To confirm this, we ran the same experiment with this *cvc5* rewrite disabled and found that *libspectre* finds 26 **sat** instances and 1361 **unsat** instances. Unfortunately, disabling this *cvc5* rewrite has a mixed effect overall, so we ultimately decided not to disable it for our other experiments. However, it does suggest that there is room for improvement in *cvc5*’s rewrite heuristics.

The results for disabling the inequality solver are in Fig. 9. We only ran this experiment on *libspectre* and *libtasn1*, as only these benchmarks produced assertions with string inequalities. For *libspectre*, this involved using a different initial input from the previous experiments to guide execution down a path that produced these inequalities. On average, the version with the inequality solver enabled finds 2.1% more **unsat** instances compared to the version without it.

7 Discussion and Future work

This work sheds light on the current state of string solvers and their usefulness in symbolic execution. While our results demonstrate much potential, there is also much room for improvement. As seen in Fig. 7, the string side of hybrid SYMCC-STR can only generate new inputs early on, and then mostly times out as the set of assertions grows. To make use of the string solver for whole programs, we need solvers that can handle the number and complexity of assertions generated by symbolic execution. To help with this goal, we have created a new set of string benchmarks generated by SYMCC-STR that we will submit for inclusion in SMT-LIB. These include many benchmarks that are highly challenging for current solvers [1]. We hope this benchmark set will motivate improvements in string solvers that will ultimately benefit the application of symbolic execution.

Our work also has already identified several promising directions for string solver developers. In particular we identified a set of conditional rewrites and a

custom string inequality solver that are both useful for symbolic execution and that could be implemented directly within string solvers.

Another observation regards *incremental solving*. Recall that each time a branch is encountered, its condition is negated and solved. In practice, this process involves repeatedly pushing, solving, and popping these negated branch conditions—i.e., incremental solving. During our experimentation, we noticed that `cvc5` would sometimes time out on incremental assertions, but could quickly solve these same assertions non-incrementally. As a workaround, we ended up frequently resetting the solver instead of solving incrementally. However, this repeated resetting of the solver significantly increases the runtime; it would be better to improve the incremental mode of the solver to reduce such mismatches between incremental and non-incremental solving.

A final observation regards conversions. As discussed in §5.1, conversions often lead to prohibitively slow performance in SYMCC-STR. This suggests that improvements to how such conversions are handled could have a significant impact on the usefulness of solvers for symbolic execution.

While there is clearly a need for improving string solvers, future work should also explore ways to reduce the burden on the string solver within SYMCC-STR. For one, as discussed in §5.1, we would like to explore more optimal data representation strategies (e.g., using taint tracking to determine, for each symbolic variable, the representation choice that will minimize conversions). Additionally, we would like to improve our solver selection strategy. Currently, we query the bitvector solver first, and if that returns `unsat`, we always query the string solver after (§4.2). However, static analysis could be used to determine if the string solver is unlikely to produce a useful result for a certain branch, allowing its invocation to be skipped. If many queries to the string solver can be skipped, then hybrid SYMCC-STR’s runtime could be greatly reduced.

8 Conclusion

We have presented SYMCC-STR, a concolic execution engine for string programs that builds on top of SYMCC. SYMCC-STR uses a novel hybrid bitvector-string data representation in order to achieve the best of both worlds: the strong performance of the bitvector solver, with the additional expressivity of the string solver. With this hybrid representation and other optimizations, we achieve a 318% increase in branch coverage on real-world benchmarks over pure-bv SYMCC-STR.

Data-Availability Statement. Our artifact is available on Zenodo [2]. It contains the SYMCC-STR source code, experimental benchmarks and scripts, and instructions on regenerating the data, statistics, and graphs from the experiments.

References

1. <https://doi.org/10.5281/zenodo.18262988>
2. <https://doi.org/10.5281/zenodo.18225511>

3. Amadini, R.: A survey on string constraint solving. *ACM Computing Surveys (CSUR)* **55**(1), 1–38 (2021)
4. Amadini, R.: A survey on string constraint solving. *ACM Comput. Surv.* **55**(2), 16:1–16:38 (2023). <https://doi.org/10.1145/3484198>, <https://doi.org/10.1145/3484198>
5. Amadini, R., Andrion, M., Gange, G., Schachte, P., Søndergaard, H., Stuckey, P.J.: Constraint programming for dynamic symbolic execution of javascript. In: Rousseau, L., Stergiou, K. (eds.) *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings. Lecture Notes in Computer Science*, vol. 11494, pp. 1–19. Springer (2019). https://doi.org/10.1007/978-3-030-19212-9_1, https://doi.org/10.1007/978-3-030-19212-9_1
6. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K.S., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: Bjørner, N.S., Gurfinkel, A. (eds.) *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*. pp. 1–9. IEEE (2018). <https://doi.org/10.23919/FMCAD.2018.8602994>, <https://doi.org/10.23919/FMCAD.2018.8602994>
7. Ball, T., Daniel, J.: Deconstructing dynamic symbolic execution. In: *Dependable Software Systems Engineering*, pp. 26–41. IOS Press (2015)
8. Bang, L., Aydin, A., Phan, Q., Pasareanu, C.S., Bultan, T.: String analysis for side channels with segmented oracles. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. pp. 193–204. ACM (2016). <https://doi.org/10.1145/2950290.2950362>, <https://doi.org/10.1145/2950290.2950362>
9. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A versatile and industrial-strength smt solver. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. pp. 415–442. Springer (2022)
10. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard – version 2.0. In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (SMT '10)* (Jul 2010), <http://theory.stanford.edu/~barrett/pubs/BST10.pdf>, edinburgh, Scotland
11. Bjørner, N.S., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. Lecture Notes in Computer Science*, vol. 5505, pp. 307–321. Springer (2009). https://doi.org/10.1007/978-3-642-00768-2_27, https://doi.org/10.1007/978-3-642-00768-2_27
12. Borzacchiello, L., Coppa, E., Cono D’Elia, D., Demetrescu, C.: Memory models in symbolic execution: key ideas and new thoughts. *Software Testing, Verification and Reliability* **29**(8), e1722 (2019)
13. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: *OSDI*. vol. 8, pp. 209–224 (2008)
14. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: *2012 IEEE Symposium on Security and Privacy*. pp. 380–394. IEEE (2012)

15. Chen, T., Flores-Lamas, A., Hague, M., Han, Z., Hu, D., Kan, S., Lin, A.W., Rümmer, P., Wu, Z.: Solving string constraints with regex-dependent functions through transducers with priorities and variables. *Proc. ACM Program. Lang.* **6**(POPL), 1–31 (2022). <https://doi.org/10.1145/3498707>, <https://doi.org/10.1145/3498707>
16. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.* **3**(POPL), 49:1–49:30 (2019). <https://doi.org/10.1145/3290362>, <https://doi.org/10.1145/3290362>
17. Chen, Y., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síc, J.: Z3-noodler: An automata-based string solver. In: Finkbeiner, B., Kovács, L. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part I. Lecture Notes in Computer Science*, vol. 14570, pp. 24–33. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_2, https://doi.org/10.1007/978-3-031-57246-3_2
18. Chipounov, V., Georgescu, V., Zamfir, C., Candea, G.: Selective symbolic execution. In: *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)* (2009)
19. David, R., Bardin, S., Ta, T.D., Mounier, L., Feist, J., Potet, M.L., Marion, J.Y.: Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. vol. 1, pp. 653–656. IEEE (2016)
20. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. pp. 337–340. Springer (2008)
21. Ghosh, I., Shafiei, N., Li, G., Chiang, W.: JST: an automatic test generation tool for industrial java applications with strings. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18–26, 2013*. pp. 992–1001. IEEE Computer Society (2013). <https://doi.org/10.1109/ICSE.2013.6606649>, <https://doi.org/10.1109/ICSE.2013.6606649>
22. Godefroid, P., Levin, M.Y., Molnar, D.A., et al.: Automated whitebox fuzz testing. In: *NDSS*. vol. 8, pp. 151–166 (2008)
23. Grossman, J.: *XSS attacks: cross site scripting exploits and defense*. Syngress (2007)
24. Hadarean, L., Barrett, C., Jovanović, D., Tinelli, C., Bansal, K.: A tale of two solvers: Eager and lazy approaches to bit-vectors. In: Biere, A., Bloem, R. (eds.) *Proceedings of the 26th International Conference on Computer Aided Verification (CAV '14). Lecture Notes in Computer Science*, vol. 8559, pp. 680–695. Springer (Jul 2014), <http://theory.stanford.edu/~barrett/pubs/HBJ+14.pdf>, vienna, Austria
25. Halfond, W.G.J., Viegas, J., Orso, A.: A classification of SQL injection attacks and countermeasures. In: Jr., S.T.R. (ed.) *2006 IEEE International Symposium on Secure Software Engineering, ISSSE 2006, Arlington, VA, USA, March 16–17, 2006* (2006)
26. Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: Hind, M., Diwan, A. (eds.) *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*

- 2009, Dublin, Ireland, June 15-21, 2009. pp. 188–198. ACM (2009). <https://doi.org/10.1145/1542476.1542498>, <https://doi.org/10.1145/1542476.1542498>
27. Kausler, S., Sherman, E.: Evaluation of string constraint solvers in the context of symbolic execution. In: Crnkovic, I., Chechik, M., Grünbacher, P. (eds.) ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014. pp. 259–270. ACM (2014). <https://doi.org/10.1145/2642937.2643003>, <https://doi.org/10.1145/2642937.2643003>
 28. Kiezun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **21**(4), 1–28 (2013)
 29. Li, G., Ghosh, I.: PASS: string solving with parameterized array and interval automaton. In: Bertacco, V., Legay, A. (eds.) Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings. Lecture Notes in Computer Science, vol. 8244, pp. 15–31. Springer (2013). https://doi.org/10.1007/978-3-319-03077-7_2, https://doi.org/10.1007/978-3-319-03077-7_2
 30. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A dpll (t) theory solver for a theory of strings and regular expressions. In: Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26. pp. 646–662. Springer (2014)
 31. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C., Deters, M.: An efficient smt solver for string constraints. *Formal Methods in System Design* **48**, 206–234 (2016)
 32. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., Barrett, C.: A decision procedure for regular membership and length constraints over unbounded strings. In: International Symposium on Frontiers of Combining Systems. pp. 135–150. Springer (2015)
 33. Loring, B., Mitchell, D., Kinder, J.: Expose: practical symbolic execution of standalone javascript. In: Erdogmus, H., Havelund, K. (eds.) Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017. pp. 196–199. ACM (2017). <https://doi.org/10.1145/3092282.3092295>, <https://doi.org/10.1145/3092282.3092295>
 34. Lotz, K., Goel, A., Dutertre, B., Kiesl-Reiter, B., Kong, S., Nowotka, D.: Solving string constraints with concatenation using SAT. In: Narodytska, N., Rümler, P. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024. pp. 29–38. IEEE (2024). https://doi.org/10.34727/2024/ISBN.978-3-85448-065-5_9, https://doi.org/10.34727/2024/ISBN.978-3-85448-065-5_9
 35. Mann, M., Wilson, A., Zohar, Y., Stuntz, L., Irfan, A., Brown, K., Donovan, C., Guman, A., Tinelli, C., Barrett, C.W.: Smt-switch: A solver-agnostic C++ API for SMT solving. In: Li, C., Manyà, F. (eds.) Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12831, pp. 377–386. Springer (2021). https://doi.org/10.1007/978-3-030-80223-3_26, https://doi.org/10.1007/978-3-030-80223-3_26

36. Pasareanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehltitz, P.C., Rungta, N.: Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.* **20**(3), 391–425 (2013). <https://doi.org/10.1007/S10515-013-0122-2>, <https://doi.org/10.1007/s10515-013-0122-2>
37. Poeplau, S., A.Francillon: Symqemu: Compilation-based symbolic execution for binaries. In: NDSS 2021, Network and Distributed System Security Symposium. Internet Society (2021)
38. Poeplau, S., Francillon, A.: Symbolic execution with {SymCC}: Don't interpret, compile! In: 29th USENIX Security Symposium (USENIX Security 20). pp. 181–198 (2020)
39. Redelinghuys, G., Visser, W., Geldenhuys, J.: Symbolic execution of programs with strings. In: Kroeze, J.H., de Villiers, R. (eds.) 2012 South African Institute of Computer Scientists and Information Technologists Conference, SAICSIT '12, Pretoria, South Africa, October 1-3, 2012. pp. 139–148. ACM (2012). <https://doi.org/10.1145/2389836.2389853>, <https://doi.org/10.1145/2389836.2389853>
40. Reynolds, A., Nötzli, A., Barrett, C.W., Tinelli, C.: High-level abstractions for simplifying extended string constraints in SMT. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562, pp. 23–42. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_2, https://doi.org/10.1007/978-3-030-25543-5_2
41. Reynolds, A., Nötzli, A., Barrett, C.W., Tinelli, C.: A decision procedure for string to code point conversion. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12166, pp. 218–237. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_13, https://doi.org/10.1007/978-3-030-51074-9_13
42. Reynolds, A., Nötzli, A., Barrett, C.W., Tinelli, C.: Reductions for strings and regular expressions revisited. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020. pp. 225–235. IEEE (2020). https://doi.org/10.34727/2020/ISBN.978-3-85448-042-6_30, https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_30
43. Reynolds, A., Woo, M., Barrett, C.W., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10427, pp. 453–474. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_24, https://doi.org/10.1007/978-3-319-63390-9_24
44. Rungta, N.: A billion SMT queries a day (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 3–18. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_1, https://doi.org/10.1007/978-3-031-13185-1_1
45. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: 31st IEEE Symposium on Security and Privacy, SP 2010, 16-19 May 2010, Berkeley/Oakland, California, USA. pp. 513–528. IEEE Computer Society (2010). <https://doi.org/10.1109/SP.2010.38>, <https://doi.org/10.1109/SP.2010.38>

46. Scott, J.D., Flener, P., Pearson, J.: Bounded strings for constraint programming. In: 25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2013, Herndon, VA, USA, November 4-6, 2013. pp. 1036–1043. IEEE Computer Society (2013). <https://doi.org/10.1109/ICTAI.2013.155>, <https://doi.org/10.1109/ICTAI.2013.155>
47. Serebryany, K.: {OSS-Fuzz}-google's continuous fuzzing service for open source software (2017)
48. Shannon, D., Ghosh, I., Rajan, S., Khurshid, S.: Efficient symbolic execution of strings for validating web applications. In: Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009). pp. 22–26 (2009)
49. Shannon, D., Hajra, S., Lee, A., Zhan, D., Khurshid, S.: Abstracting symbolic execution with string analysis. In: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007). pp. 13–22. IEEE (2007)
50. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al.: Sok:(state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE symposium on security and privacy (SP). pp. 138–157. IEEE (2016)
51. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016. The Internet Society (2016), <http://wp.internet-society.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf>
52. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: a z3-based string solver for web application analysis. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013. pp. 114–124. ACM (2013). <https://doi.org/10.1145/2491411.2491456>, <https://doi.org/10.1145/2491411.2491456>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder





Same Engine, Multiple Gears: Parallelizing Fixpoint Iteration at Different Granularities

Ali Rasim Kocal¹ , Michael Schwarz² ,
Simmo Saan³ , and Helmut Seidl¹

¹ Technical University of Munich, Garching, Germany

² National University of Singapore, Singapore

³ University of Tartu, Estonia

{ali.rasim.kocal, helmut.seidl}@tum.de

m.schwarz@nus.edu.sg, simmo.saan@ut.ee

Abstract. Fixpoint iteration constitutes the algorithmic core of static analyzers. Parallelizing the fixpoint engine can significantly reduce analysis times. Previous approaches typically fix the granularity of tasks upfront, e.g., at the level of program threads or procedures — yielding an engine permanently stuck in one gear. Instead, we propose to parallelize a generic fixpoint engine in a way that is parametric in the task granularity — meaning that our engine can be run in different gears. We build on the top-down solver **TD**, extended with support for mixed-flow sensitivity, and realize two competing philosophies for parallelization, both building on a task pool that schedules tasks to a fixed number of workers. The nature of tasks differs between the philosophies. In the *immediate* approach, all tasks access a single thread-safe hash table maintaining solver state, while in the *independent* approach, each task has its own state and exchanges data with other tasks via a *publish/subscribe* data structure. We have equipped the fixpoint engine of the static analysis framework **GOBLINT** with implementations following both philosophies and report on our results for large real-world programs.

Keywords: abstract interpretation · static analysis · fixpoint iteration · parallelization.

1 Introduction

Static analysis based on abstract interpretation tries to infer program invariants without running the program. For that, candidate invariants are represented by *abstract values* from some bounded lattice (the *domain*) where the lattice operations *meet* and *join* correspond to logical conjunction and disjunction respectively. The analysis of the program then gives rise to a system of equations over the domain whose solutions correspond to inductive invariants of the program. The design of such an analysis consists of defining and implementing the abstract domains and the corresponding abstract transfer functions corresponding to edges in control-flow graphs, as well as specifying the relationships among abstract values of, e.g., program points (in context) or global variables.

Ideally, the designer of such an analysis should concentrate on these matters without simultaneously being concerned with the algorithmic details of how a solution of the system of equations is computed. A separation of the design of the analysis and the implementation of the fixpoint engine is important – particularly when the fixpoint engine is intricate. As a result, fixpoint engines such as [31, 42] are ideally implemented in an *application-independent* manner to enable re-use. **TD** [31] is one such fixpoint engine. It is broadly applicable as it makes no assumptions on the finiteness of the set of unknowns, requires no a priori knowledge about dependencies between variables of the equation systems (which we call *unknowns* to distinguish them from program variables), and dynamically detects points where to apply widening/narrowing. Moreover, it supports *mixed-flow sensitive* analyses where some unknowns are analyzed flow-insensitively while the values of others are determined flow-sensitively [38].

While analyzers such as **ASTRÉE** [21] and **PIKOS** [15] have been parallelized, many others, such as **GOBLINT** [43], are sequential programs and thus do not exploit multiple cores for speeding up the fixpoint computation. Here, we aim to provide *generic* parallelization strategies that do not pose restrictions on the programs that can be analyzed, and can easily be adapted to a variety of analyses, in particular, mixed-flow sensitive analyses, and support multiple *granularities* of tasks. Rather than supporting only a predefined set of granularities, tasks are identified by *root* unknowns. For each root x , a separate instance of the solver is created for locally solving for x by exploring the equation system in a demand-driven way. To realize a parallelization of the analysis, e.g., at the level of program threads, the endpoints of created program threads are selected as roots. For parallelizing the analysis of sequential code, potential choices are the endpoints of procedures or branches of specific *if-else* statements. To minimize the synchronization overhead at accesses to shared data, distinct roots ideally should result in the exploration of disjoint sets of unknowns. In practice, though, this can hardly be guaranteed. We propose two approaches for dealing with the interaction between distinct solver tasks:

- The *immediate* approach maintains all solver data, grouped by the unknowns to which they refer, within one shared thread-safe data structure, which guarantees *transactional* accesses by distinct tasks to individual unknowns.
- The *independent* approach, on the other hand, maintains multiple local copies of solver data and introduces a *publish-subscribe* mechanism to share information between different tasks.

The rest of this paper is structured as follows: Section 2 describes by example how *side-effecting equation systems* can be used to formalize the analysis of multi-threaded C programs. Section 3 introduces the improved version **TLTD** (toplevel **TD**) of **TD** that lays the groundwork for our parallelization. Section 4 then presents our two approaches for parallelizing **TLTD**, which are experimentally evaluated in Section 5. After discussing related work on parallel solvers in Section 6, we summarize our results in Section 7.

2 Background

The basic setup of thread-modular analysis, as considered, e.g., in [29, 30], aims at inferring a single invariant for each program point. To that end, for each program point v , an *unknown* is introduced taking values from some bounded lattice $\mathbb{D}$ of candidate invariants. A right-hand side function f_v is provided for every program point v to indicate how the invariant for v can be obtained from the invariants for other program points, while taking the control-flow of the program into account. Such unknowns are called *locals*. In a refined setting, multiple invariants per program point may be computed, differentiated, e.g., by calling-contexts or thread *ids* (or both). In that case, multiple locals may be introduced for each program point, e.g., the unknown $\langle v, c \rangle$ for program point v in context c . To abstract thread interaction via shared data structures, a dedicated further set of *globals* is introduced. These are unknowns whose values represent invariants of shared data that are meant to hold throughout the whole program execution. Their values are therefore accumulated *flow-insensitively*.

```

1  int g;
2  void* foo(void* a) {
3      g = (int) a;
4      return NULL;
5  }
6  int main() {
7      pthread_t t1; int a;
8      g = 0;
9      pthread_create(&t1, NULL, foo, (void *) 42);
10     a = g;
11     a = a+1;
12     return a;
13 }
```

Fig. 1: A multi-threaded C program

Consider the multi-threaded C program in Figure 1, which serves as our running example. Assume that we are going to perform *interval* analysis [8] in a thread-modular way. Interval analysis for the program in Figure 1 attempts to infer tight lower and upper bounds for the values of a and g . Since the variable g is shared, we introduce a global unknown for g , for convenience, with the same name. As a bounded lattice of candidate invariants for g , we use the set $\mathbb{I}$ of intervals (extended with $\emptyset$) equipped with the natural ordering.

The set of locals in our examples consists of the set of program points. For technical reasons, we consider *start points* of called procedures as globals, since these do not receive their values via right-hand sides, but via contributions from call-sites dispersed all over the program. In general, interval analysis will maintain a separate interval for each local program variable. In our minimalistic

example, however, there is just a single local variable a in both functions `main` and `foo`. The candidate invariants for locals may thus be taken from the same domain $\mathbb{I}$ as for globals in this case. In the following, we assume that locals are named according to lines in the program text of Figure 1.

3 The Solver TLTD

Let $\mathcal{X}$ denote the set of all unknowns. A run of the analysis consists of repeatedly evaluating right-hand sides until a valuation of unknowns $\sigma : \mathcal{X} \rightarrow \mathbb{D}$ is found so that each right-hand side is accounted for. Specifically, this means that re-evaluating each right-hand side only produces results already subsumed by σ .

In this section, we describe a version **TLTD** of the top-down solver **TD** [31], which has a modified organization of its data structures and an additional top-level workset. This modification, while also beneficial on its own, is the groundwork for our parallelization. The implementation is presented in Figures 2, 3, and 4, where the parts differing from **TD** are highlighted, apart from those related to the modification of data structures.

TD solves for unknowns recursively as soon as they are encountered during the analysis. In contrast, **TLTD** introduces a main loop (Figure 2) that calls the function `iterate` for each unknown x in the workset `work` to determine the value for x . Initially, `work` is assumed to contain the *unknowns of interest*, typically, the endpoint of `main` ($\langle 13 \rangle$ in our example), which triggers the iteration on all possibly *influencing* unknowns. The set `work` may *dynamically* receive further root unknowns during the analysis, enabling us to solve for unknowns in parallel.

```

46  while not (Set.is_empty !work) do (
47    iterate (Set.choose !work)
48  ) done

```

Fig. 2: The top-level loop of **TLTD**

The core of the **TLTD** algorithm is the `iterate` function (Figure 3). When called for an unknown x , `iterate` repeatedly evaluates the right-hand side for x (line 26), until a fixpoint is reached. The right-hand side provided by the call `eq x` is a higher-order function that formalizes how the abstract value of x can be determined. It is parametrized by the three functional arguments `get`, `set` and `demand`, which represent the *interface* to the fixpoint engine. The first functional argument `get` is used to access the values of other unknowns.

Example 1. Consider the assignment `a = a + 1`; in line 11 of Figure 1. Then the right-hand side of the unknown $\langle 12 \rangle$ following that line is given by

$$\text{eq } \langle 12 \rangle = \text{fun get set demand} \rightarrow (\text{get } \langle 11 \rangle) +^\sharp [1, 1] \quad (1)$$

where the $+^\sharp$ is the abstract version of the programming language operator “+” for the domain $\mathbb{I}$. $\square$

```

1  let rec iterate x =
2    let rec query y =
3      let dy = find data y in
4      if (not (dy.called) && not (is_global y)) then (
5        iterate y
6      );
7      dy.influences <- (Set.add x dy.influences);
8      dy.value in
9    let side g contribution =
10     let dg = find data g in
11     dg.stable <- true;
12     let new_value = Domain.join dg.value contribution in
13     if not (Domain.leq new_value dg.value) then (
14       dg.value <- new_value;
15       destabilize g;
16     ) in
17   let promote y =
18     let dy = find data y in
19     dy.toplevel <- true;
20     work := Set.add y !work in
21
22   let dx = find data x in
23   dx.called <- true;
24   while not dx.stable do (
25     dx.stable <- true;
26     let new_value = eq x query side promote in
27     if not (Domain.leq dx.value new_value) then (
28       dx.value <- new_value;
29       destabilize x;
30     );
31   ) done;
32   if dx.toplevel then (work := Set.remove x !work);
33   dx.called <- false

```

Fig. 3: The core fixpoint iterator of **TLTD**.

The values for program globals are analyzed *flow-insensitively* and represented by *global* unknowns (*globals*) in $\mathcal{X}$. Globals do not have right-hand sides on their own, but receive contributions from the right-hand sides of locals. The second argument **set**, therefore, provides a means to produce such contributions (called *side-effects*).

Example 2. Consider the assignment $\mathbf{g} = (\text{int}) \ \mathbf{a}$; in line 3. Then

$$\text{eq } \langle 4 \rangle = \mathbf{fun} \ \text{get set demand} \rightarrow \mathbf{let} \ d = \text{get } \langle 3 \rangle \ \mathbf{in} \quad (2)$$

$$\quad \text{set } g \ d;$$

$$\quad d$$

Here, first the value d of $\mathbf{a}$ is obtained via **get**, which also happens to be the value to be propagated to the unknown $\langle 4 \rangle$. Additionally, however, a contribution is provided to the global g by means of the call **set** $g \ d$. $\square$

The third argument **demand** is introduced by **TLTD**. It can be called on unknowns, of which the values must be calculated, e.g., for their side-effects, but are not used directly. The following example illustrates a potential usage.

Example 3. Consider the right-hand side for the program point 10, following the creation of a new thread in line 9. The created thread will concurrently execute the function **foo** for input 42. The corresponding equation therefore contributes the interval $[42, 42]$ as an abstraction of 42 to the start point of **foo**. The result of the evaluation of **foo** is not required to determine the value for the next program point 10. Nevertheless, the body of **foo** must be analyzed, as it might affect global unknowns through **set**. Therefore, **demand** is called for the endpoint $\langle 5 \rangle$ of **foo**. Accordingly, we define

$$\text{eq } \langle 10 \rangle = \mathbf{fun} \ \text{get set demand} \rightarrow \mathbf{let} \ d = \text{get } \langle 9 \rangle \ \mathbf{in} \quad (3)$$

$$\quad \text{set } \langle 3 \rangle \ [42, 42];$$

$$\quad \text{demand } \langle 5 \rangle;$$

$$\quad d \quad \square$$

To summarize, the three argument functions **get**, **set** and **demand** represent the generic *interface* which allows **TLTD** to interact with the evaluation of right-hand sides in the following way:

- The functional parameter **get** receives the local solver function **query**. That function, when called for some unknown y , ultimately returns the value of y as stored in **data**. Before that, however, an up-to-date value of y is calculated by calling **iterate** whenever an iteration on y has not already been started. To check the latter, **data** for y maintains a flag **called** which is set by the function **iterate** before it enters the *while* loop to perform fixpoint iteration on y . Once iteration for y has terminated, the dependence of x on y is recorded by inserting x into the set attribute **influences** of y in **data**, and the latest value of y is returned.

- The functional parameter **set** receives the local function **side**, which has arguments a global g and a contribution to g . First, the attribute **stable** of g in **data** is set to **true** to include g in the result set of unknowns. If the new contribution is subsumed by the old value of g , the solver proceeds. Otherwise, the value of g in **data** is updated to the join of old and new values, and **destabilize** is called to inform all other unknowns that directly or indirectly depend on g , as their current values may be outdated.
- Finally, the solver function **promote** is provided for the functional parameter demand. When called for an unknown y , **promote** adds y to the top-level workset and records in **data** that y has now been raised to the top-level. Note that no influence is recorded for y .

When called on a hashmap **data** and an unknown x , the helper function **find** – which is left out for brevity – looks up a record for x in **data**, creates one if there is none present and returns a reference to the record consisting of the fields accessed in Figure 3. This is different from **TD** as presented in [31], which organizes the data in individual hashmaps for each field.

In detail, the iteration on x performed by the function **iterate** proceeds as follows: First, the flag **called** for x in **data** is set to **true**. The iteration on x is repeated until the value of x as stored in **data** for x does not change anymore, i.e., *stabilizes*. To indicate that fact, the record **data** for x has an entry **stable**, which is optimistically set to **true** before the *while* loop. Whenever the evaluation of the right-hand side for x provides a value that is not subsumed by the old value, the entry **value** of the record **data** x is updated to the join of the old value for x with the new value. Furthermore, for all other unknowns that are directly or indirectly influenced by x , the entries **stable** are set to **false** by calling the auxiliary function **destabilize**. The function *destabilize*, as presented in Figure 4, additionally checks whether a destabilized unknown y is top-level. If this is the case, it is added back to the toplevel workset **work** for re-iteration, ensuring all influences are accounted for even when y is not queried again. Finally, assume that the unknown x has *not* been destabilized – either during the last evaluation of its right-hand side or by the last call to **destabilize** after a modification of the entry **value** of x . Then the iteration has terminated. In case x has been marked as **toplevel**, x is also removed from **work**. In the end, the entry **called** in **data** x is reset to **false**.

Example 4. In our example, the global g receives a contribution $[0, 0]$ via the right-hand side of program point $\langle 9 \rangle$ in **main**. That value will be read when evaluating the right-hand side for $\langle 11 \rangle$, and will be incremented in the right-hand side for $\langle 12 \rangle$ and thus provides the first value for the endpoint $\langle 13 \rangle$ of **main**. In the right-hand side for program point $\langle 10 \rangle$, however, a contribution of $[42, 42]$ occurs to the start point $\langle 3 \rangle$ of function **foo**, while in the same right-hand side the endpoint $\langle 5 \rangle$ of **foo** is turned into a top-level unknown and inserted into **work**. The subsequent iteration on $\langle 5 \rangle$ results in a further contribution of $[42, 42]$ to g . By joining this new contribution with the old value of g , the solver obtains the interval $[0, 42]$ for g . The subsequent call to **destabilize** will put the endpoint $\langle 13 \rangle$ of **main** back into the workset **work**. In a second go on the

```

34  and destabilize x =
35    let dx = find data x in
36    let influences = dx.influences in
37    dx.influences <- Set.empty;
38    Set.iter (fun y ->
39      let dy = find data y in
40      dy.stable <- false;
41      if dy.toplevel then (
42        work := Set.add y !work
43      );
44      destabilize y
45    ) influences

```

Fig. 4: The auxiliary function `destabilize`

body of `main`, the solver will recompute the values for $\langle 11 \rangle$ and $\langle 12 \rangle$ to $[0, 42]$ and $[1, 43]$, respectively, which then is also the final value for the endpoint of `main`. $\square$

The changes of **TLTD** compared to **TD** prepare the ground for parallelization:

- Fusing several data structures (as maintained by **TD**) into one reduces the number of possibly expensive accesses to large data structures, which later may be shared between multiple tasks.
- Introducing a top-level workset provides a mechanism to replace the recursive descent into unknowns with a *delayed* or, as described in Section 4, *parallel* execution.

The first modification does not change the semantics of the solver compared to the original **TD** (a formal correctness proof of the original **TD** without side effects is given at [36, 37]). The second modification assures that each delayed unknown x once added to the top-level worklist, will be re-entered into that worklist whenever it is destabilized. In this way, x is guaranteed to be iterated upon until stabilization.

The version of **TLTD** as implemented in GOBLINT additionally supports *widening* and *narrowing* [8, 9]. This addition is orthogonal to its parallelization. Therefore, it has been omitted here.

4 Parallelizing TLTD

Task-pool-based parallelization is a common approach, in which a number of *tasks* are scheduled on a generally lower number of processors. It has, e.g., been used for the parallelization of *modular* analyses [41]. This approach is suitable when the analysis consists of sufficiently large modules, such as threads or functions. However, for solving systems of equations as described in Section 1, the unknowns, and therefore tasks, would correspond not to large components but

to individual program points. With such tiny components in large numbers, creating one solver thread for every unknown is impractical. Our idea is to use only the top-level workset of **TLTD** as our task pool for parallelization. Tasks are identified dynamically by root unknowns triggering the *recursive exploration* of larger parts of the equation system, and are mapped to a fixed set of workers by the task-pool. This enables parallelization without a priori knowledge of the equation system. However, the partitioning into solver tasks takes place in a non-deterministic fashion, and the right-hand sides may be evaluated in different orders in multiple solver runs. In the presence of *widening and narrowing*, this can lead to solutions with varying precision. In our running example, the top-level workset is initialized with the endpoint of `main` and the endpoint of `foo` later makes it into the top-level workset. Calling `iterate` for these in different tasks will trigger one task to explore the unknowns from the body of `main` and further unknowns from the body of `foo`. We propose two different approaches to how the interaction between multiple tasks is organized.

- The *immediate* approach maintains all solver data within one shared thread-safe data structure, which guarantees *transactional* accesses by distinct workers to individual unknowns.
- The *independent* approach, on the other hand, maintains multiple local copies of solver data and introduces a *publish-subscribe* mechanism to share information between tasks.

The *immediate* approach assures that at any point in time, at most one worker is iterating on any given unknown. In this way, spurious computation is ruled out – at the price of synchronization at every data access. The *independent* approach does not attempt to exclude duplication of work. Instead, it reduces the need for synchronization to the accesses to the publish-subscribe mechanism. In detail, these approaches work as follows.

4.1 The *Immediate* Approach

The *immediate* approach relies on a thread-safe implementation of the data structure `data`. To ensure that a worker operating on an unknown x only writes consistent results to `data`, i.e., results that are based on the latest values in `data`, we use a *transactional* model (see, e.g., [11]). Thus, for any operation on x , such as `iterate`, the data is read at the beginning of the operation.

If the data changes before results are saved in the shared data, we repeat the whole operation with newly read data. In the implementation, this is achieved by replacing the records in `data` with atomic records on which we can apply *compare-and-swap* operations (CAS). The CAS operation sets the new value of a record only if the last known value, as provided by a second parameter, is unchanged. Note that this approach enables a *lock-free* implementation thanks to the fact that **TLTD** keeps a single reference for all data related to an unknown. We use a hashmap with buckets implemented as *Michael-Scott* queues [20]. In our application, the atomic wrapper for the data of an unknown, once initialized,

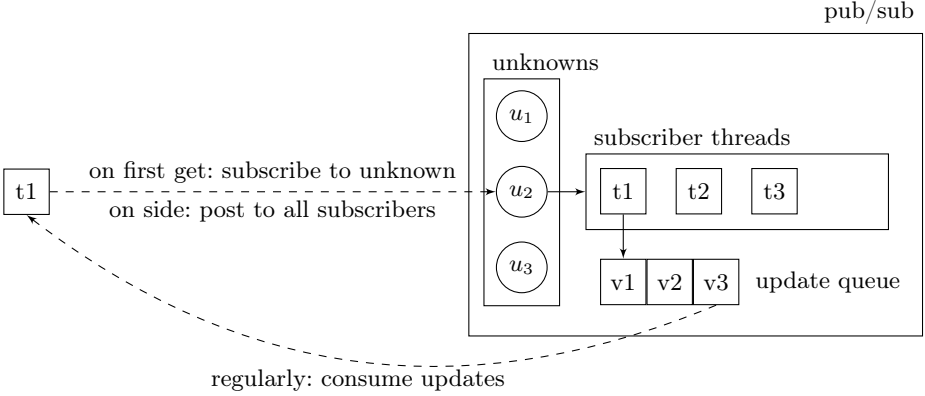


Fig. 5: Pub/Sub system for global unknown updates

never changes – only the values it refers to may be updated. Moreover, it is never removed from **data**, allowing for a simplification and efficient implementation of the hashmap.

For good performance, repeated operations due to CAS-failures must be exceedingly rare. Note that once a task t has called the function **iterate** on an unknown x , no other task will call **iterate** on x until t finishes iteration and sets **called** to **false**. Therefore, t has sole write access to the entries **called** and **value** of x , whereas other solver threads only read them. The only field that is updated frequently by other solver threads, potentially causing an undue amount of CAS failures, is **influences**. Therefore, **influences** is implemented as a *thread-safe set* and excluded from the transactional model. This design decision reduces the ratio of repeated operations due to CAS failures down to less than 0.01% on average. To maintain correctness, nonetheless, it must be ensured that every unknown y that depends on x is destabilized when x receives a new value. We achieve this by always destabilizing x *after* a new value is saved and expanding **influences** of x with y *before* the value of x is read during the iteration for y . When x receives a new value between the expansion of **influences** and the reading of the new value, this strategy can lead to premature destabilization of y and an unnecessary recalculation resulting in the same value. However, this is a rare case with minimal impact on performance.

The solver terminates once the iteration for all encountered unknowns marked as *toplevel* has terminated.

4.2 The Independent Approach

Ideally, a good selection of root-unknowns leads to recursive descent into mostly distinct parts of the equation system. Notably, this is the case when root unknowns correspond to the endpoints of program threads. Our second approach, therefore, assumes that, if at all, few local unknowns will be encountered by

multiple tasks, and their recalculation is less costly than the synchronization overhead. Thus, it lets each task operate on a *distinct copy* of the hashmap **data**. Synchronization is only necessary on the values of globals as they do not have right-hand sides, but may receive contributions from multiple locals. We use a publish/subscribe system [23] to share those contributions among tasks, as illustrated in Figure 5. Upon first read of a global g , a worker *subscribes* to the contributions for g , while each contribution to g is *published* to its subscribers' thread-safe update-queues. This is easily achieved by extending the **query** and **side** functions. Additionally, all contributions for globals are kept, so that new subscribers can receive the most up-to-date values as well. Each task regularly (e.g., after the evaluation of every right-hand side) consumes all new updates to the globals it has subscribed to. It does so by updating their values in its local copy of **data** and destabilizing influenced unknowns if necessary. Further, tasks are revived if a global unknown that they have subscribed to receives an update after the task has terminated, possibly invalidating its former results. This guarantees that all the relevant updates to the globals are taken into account by every solver task. The solver then terminates when all tasks have terminated, and all update-queues are fully processed. Apart from the handling of the globals, the termination criteria ensuring correctness are the same as in **TLTD**.

5 Experimental Results

We have implemented both philosophies for parallelizing **TLTD** in OCAML 5.3 using the module **DOMAINS** for obtaining true concurrency [35]. The improved *sequential* solver **TLTD** from Section 2 (extended with widening and narrowing) can be recovered by running the *immediate* version of **TD** with a single worker thread. These solvers have been integrated into the open-source static analyzer **GOBLINT** for multi-threaded C.

The downside of parallelization is that, as the *running time* and *precision* of the result may depend on the *order* in which right-hand sides are evaluated, the behavior may differ from one parallel run to another, though all returned results are guaranteed to be sound.

Our goal, therefore, is to answer the following Research Questions.

- RQ1.** What is the impact of the improvements from Section 2 already on the *sequential* performance of **TLTD**?
- RQ2.** How do the *immediate* and *independent* approaches compare to each other and scale with the number of worker threads?
- RQ3.** How does the choice of granularity, i.e., the placement of demands, affect the efficiency of the parallelization?
- RQ4.** How does the precision vary for the two approaches and between different runs?

To answer these questions, we configured **GOBLINT** to run a context-sensitive interprocedural analysis with some loop unrolling. The analyzer computes points-to information for pointers, and interval sets for *int* variables, while tracking symbolic thread *ids* [30] and sets of definitely held mutexes.

We have collected benchmark programs from the *Concurrency* suite of SV-Comp [5] and the Concrat [13] benchmarks. We have selected the non-trivial programs from these suites by filtering out input programs, where the unmodified **TD** has terminated in under five seconds. We have extended the set of benchmarks with four open-source Unix utilities **aget**, **ctrace**, **knot**, and **pfscan** to measure performance on real-world applications. For practical reasons, we have run the experiments with a timeout of 20 minutes. This ruled out 24 programs where the unmodified **TD** has timed out and one further program where the *independent* solver has timed out. In two cases, the joined solutions of the solver threads from the *independent* solver were, although sound, not fixpoints. As this behavior leads to abnormal termination within the GOBLINT framework, we have excluded these programs as well.

Our final benchmark consists of 47 programs (30 SV-Comp, 13 Concrat, 4 Unix utilities) ranging from 23 to 12.5k logical lines of code. The experiments were conducted on Ubuntu 24.04.2 LTS with two Intel(R) Xeon(R) Platinum 8260 CPUs, each with 24 cores, where a single core corresponds to the computing power of a core in an ordinary consumer machine.

RQ1. We found that merging the various data structures of the original fixpoint algorithm, as provided by GOBLINT in one hashmap **data** already resulted in a measurable speedup of $1.18\times$ on average by reducing the time spent traversing and maintaining the data structures. We observed that the runtime of fixpoint iteration is dependent on the execution order. This is not surprising because if an unknown x directly or indirectly depends on another unknown y , calculating the value of y before x is faster, as x must otherwise be recalculated. Empirically, promoting the endpoints of called functions to the top-level workset at every call edge before querying their values resulted in a combined speedup in the range of $0.41\times$ to $9.49\times$ with an average of $1.85\times$. When unknowns were demanded at **pthread_create** only, the speedup was even more pronounced, ranging from $0.6\times$ to $10.42\times$ with an average of $2.25\times$. We conclude that the running time of **TD** can be improved by iterating over certain unknowns on the top-level, where the granularity has a significant impact on the overall performance.

RQ2. We evaluated the two parallelizing approaches for 1, 2, 4, and 8 workers, where demands are placed at **pthread_create**. For each benchmark program, we determine the sequence of speedups relative to the single-worker running time of each approach. Figure 6 shows these speedups. For better visualization of the results, we display the results only for a representative subset of programs by selecting every fourth program from the full benchmark suite, sorted by the single-worker running time. Tables containing all running times can be found in the Appendix of the extended version[17]. The *immediate* approach on all benchmarks shows average (maximal) speedups of $1.7\times$, $2.01\times$ and $2.68\times$ ($4.98\times$, $6.58\times$, $9.26\times$) with 2, 4 and 8 workers compared to when it is run with a single worker; the *independent* approach $1.83\times$, $2.5\times$ and $2.87\times$ ($4.25\times$, $7.71\times$, $9.5\times$) respectively. Thus, we find significant speedups consistently for both approaches, which increase

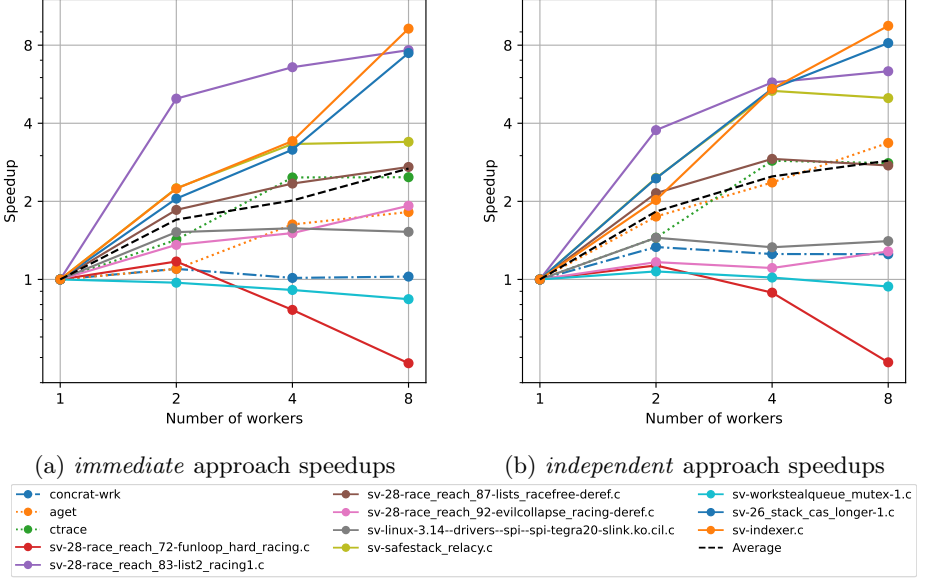
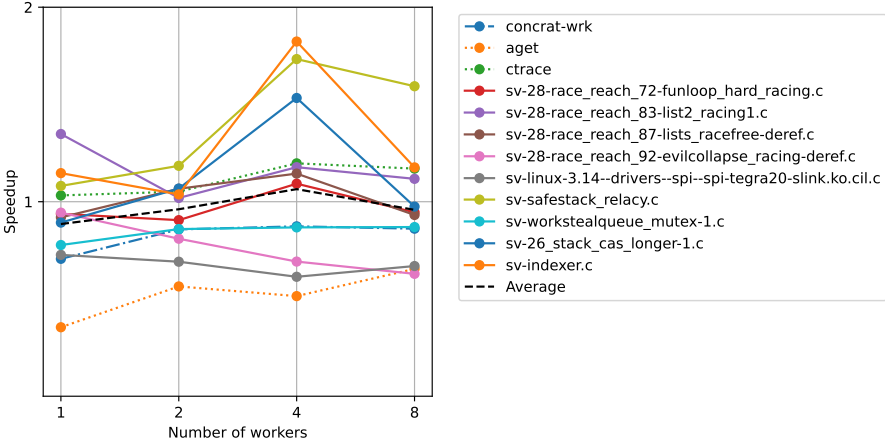


Fig. 6: Relative speedups with demand at `pthread_create`

with the number of workers. For the majority of our benchmark programs, speedups tend to diminish with more than four workers. We observe super-linear speedups in some cases, which again is explained by the variations in execution order of the right-hand sides. For a comparison of the two approaches, Figure 7 displays for every considered number of workers, the ratio of the running times of the *immediate* over the *independent* solver in the same setting. The independent and intermediate approaches have similar running-times, which almost never differ by a factor larger than 2. These differences are in either direction; neither approach dominates the other.

RQ3. To evaluate the impact of demand point selection, we have also run the *immediate* approach, while placing demands not only when the analyzed program starts a new thread via `pthread_create`, but also for the unknowns corresponding to the endpoints of every called function. Figure 8 displays these results in the same way as above. In **RQ1**, we noticed that using demand nodes sparingly, only for `pthread` creation, leads to better *single-worker* performance. In contrast, we find that the increased number of roots results in better scaling with the number of workers. This indicates that the potential for parallelism depends on the selection of demand points. In the input programs where the introduced potential for parallelism by the mentioned strategies was not large enough, speedups diminished with higher parallelization overhead incurred by larger numbers of solver threads.

RQ4. As expected, the results computed by the various solver configurations are not always identical. Still, we generally found agreement in the results for the

Fig. 7: Comparing *independent* vs. *immediate*

vast majority of unknowns (see Table 1). On average, slightly more unknowns receive more rather than less precise values compared to the solution by the unmodified **TD**. Barely any unknowns have incomparable values.

Approach	Demand	Equal	More precise	Less precise
<i>immediate</i>	pthread	0.92	0.04	0.03
<i>immediate</i>	pthread, functions	0.92	0.04	0.03
<i>independent</i>	pthread	0.89	0.06	0.05

Table 1: Precision across approaches, compared with unmodified **TD**.

Threats to Validity. A fundamental issue is how far our collection of benchmarks is representative of relevant applications. By including the non-trivial programs from already proposed suites, namely, the non-trivial concurrency benchmarks from SV-Comp and the Concreat suite, we included examples that have already been judged as reasonable by the community. By adding further multi-threaded open source Unix utilities, we tried to complement these existing suites with further commonly used programs. Indeed, our findings for programs are quite independent of their sources. When reporting our results for runs of the parallel solvers, we must be aware that, due to the operating system’s scheduling policy or the OCAML runtime’s memory management, non-deterministic behavior is to be expected. We have reported the number of single runs here to expose irregularities in performance and precision. We observe, however, that the averaged

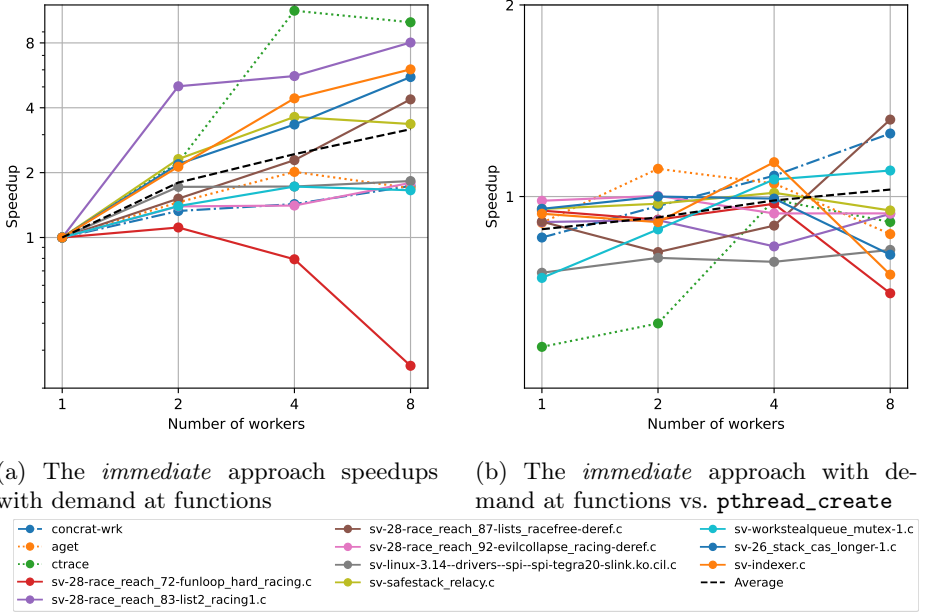


Fig. 8: Putting demand at functions

results across three runs, both for running times and the ratio of unknowns with equal precision, vary by less than 2%.

Another issue is our choice of the analysis configuration. To obtain comparable results, we did not report on results with different abstract domains or other approaches to distinguish multiple calling contexts. Our experiments, though, indicate that the outcome for these is essentially the same. Finally, it is difficult to directly compare experimental results with state-of-the-art parallelization approaches, as the underlying analyzers often have fundamental differences in their requirements and limitations. Further, the approaches we consider to have the most similarities are implemented for analyzers of different programming languages, making it unfeasible to evaluate them on the same benchmark suite.

6 Related Work

Parallelization to speed up program analysis tasks is widely applied to model checking and dynamic or static analyses. We focus on static analysis and specifically on techniques that parallelize the analyzer itself, as opposed to, e.g., *parallel portfolio* approaches where multiple (configurations of) analyzers are run in parallel [4, 18, 24], or approaches that compile analysis specifications into DATALOG programs and then use concurrent evaluation engines (e.g., [1, 14, 27, 28]).

Analyses that compute bottom-up summaries of procedures are easy to parallelize, as all leaves can be analyzed in parallel. Shi and Zhang [34] report that

many analyzers [3, 7, 19, 33, 44] make use of this idea. Based on this observation, Shi and Zhang [34] propose to increase parallelism by decomposing the analysis of a single procedure into multiple tasks.

Monniaux [22] parallelizes the ASTRÉE analyzer by computing abstract states for both branches for long-running *if-then-else* statements in parallel. In particular, he focuses on *dispatch* statements that occur in event-driven programs, where, based on the event to be processed, different potentially long-running code paths are taken. Kim et al. [15] parallelize the IKOS abstract interpreter by further weakening Bourdoncle’s weak topological order [6] to a weak *partial* order (wpo), allowing parallelization of parts of the program that are unordered w.r.t. each other. They also establish that their algorithm is deterministic: any scheduling choice consistent with the wpo will compute the same result as Bourdoncle’s original algorithm. These approaches expect the complete equation system be provided explicitly beforehand — which is not the case for context-sensitive and especially demand-driven analyses, where an implicitly provided equation system is only partly explored. Approaches requiring preprocessing of the equation system, therefore, are not applicable.

The two strands of work most closely related to ours are by Albarghouthi et al. [2] and Van Es et al. [41] (later presented in an extended version [39]), respectively. Both approaches require a set of predefined modules (functions, and additionally threads in the latter work). In our setting, however, such predefined modules are not readily available as we target the parallelization of *generic* solvers. Van Es et al. [41] treat the worklist as a task pool of components to be analyzed. The analysis of each such component yields new components to be analyzed (either because they are newly discovered or because they need to be re-analyzed as they depend on the results of the current component). In this way, communication of updates to dependencies happens only when the analysis of a component has finished, whereas in our approaches, updates to globals are communicated between tasks continuously. Albarghouthi et al. [2], on the other hand, take a more structured approach to concurrency and alternate between *map* and *reduce* phases. In the *map* phase, queries in the worklist are processed until they yield results or are blocked on dependencies. In the *reduce* phase, the state of all queries is updated, and queries that are unblocked are added back to the worklist. In our approaches, tasks do not become blocked when querying a top-level unknown that is still being computed by another task. Instead, they receive the current best knowledge about that unknown and continue their work optimistically, but they will be rescheduled if the unknown is later updated.

Rodriguez and Lhoták [26] parallelize IFDS-style analyses [25] by defining one actor per node and realizing propagation of dataflow facts as message passing between actors, delegating the propagation of messages to the runtime system. Helm et al. [12] also achieve parallelization of IFDS-style analyses, using a *semi-implicit* programming model that propagates values between *cells* where developers explicitly give dependencies between cells and mark some cells as needing to be updated non-concurrently. They require monotone right-hand sides and

bounded lattices satisfying the ascending chain condition, and do not support widening and narrowing.

Sun et al. [40] propose to execute dataflow analyses given an interprocedural control-flow graph on large compute clusters based on Apache Giraph. To this end, they devise a *distributed* fixpoint algorithm tuned to reduce the amount of communication between compute nodes by only propagating new dataflow facts, akin to *semi-naïve evaluation*. This approach is once more *not* applicable to demand-driven analyses and does not support widening and narrowing.

Sherman and Dwyer [32] propose to parallelize the analysis at a *fine-grained* level by splitting up CFGs and considering some paths in the CFG in parallel. Once more, this approach is not applicable to demand-driven analyses with dynamic dependencies between unknowns.

Dewey et al. [10] take a different approach and present the analysis of JAVASCRIPT programs as an embarrassingly parallel problem where all paths are analyzed in parallel, with a separate process controlling which abstract program states are to be merged. The motivation is that in highly dynamic languages such as JAVASCRIPT, CFGs are not readily available. This kind of forward exploration does not easily extend to the demand-driven analyses, which we study here.

7 Conclusion and Future Work

We have presented two approaches for parallelizing the generic fixpoint engine **TLTD** and have demonstrated their effectiveness for several gears, i.e., granularities, in the context of a thread-modular analysis as provided by the static analyzer GOBLINT. Already, the modifications to the original **TD** from GOBLINT in preparation for the parallelization yielded speedups up to a factor of 10, with an average speedup of $1.85\times$. When turning to parallelization, we observed *further* speedups of up to a factor of $11.29\times$ (4 workers, *immediate* approach, demand at functions), respectively $9.5\times$ (8 workers, *independent* approach, demands at thread creations), with an average speedup of $3.18\times$ (*immediate*, 8 workers) and $2.87\times$ (*independent*, 8 workers). The two parallelization approaches complement each other with no clear winner. It seems, though, that for smaller granularity, e.g., at the procedure level, the *immediate* approach is preferable.

Here, we have presented results only from experiments with two granularities. It remains for future work to explore further possibilities of demand placement, depending, e.g., on the characteristics of the program under analysis.

Acknowledgments. This work was supported in part by the European Union and the Estonian Research Council via projects PRG2764 and TEM-TA119. This work has benefited from the participation of Helmut Seidl and Michael Schwarz in the Dagstuhl Seminar 25421 *Sound Static Program Analysis in Modern Software Engineering*.

Data-Availability Statement The artifact, including source code for our implementations, benchmarks, benchmark results and the evaluation scripts, is archived and available at DOI [10.5281/zenodo.18186778](https://doi.org/10.5281/zenodo.18186778)[16].

Bibliography

- [1] Abeysinghe, S., Xhebraj, A., Rompf, T.: Flan: An expressive and efficient datalog compiler for program analysis. *Proc. ACM Program. Lang.* **8**(POPL) (2024), <https://doi.org/10.1145/3632928>
- [2] Albarghouthi, A., Kumar, R., Nori, A.V., Rajamani, S.K.: Parallelizing top-down interprocedural analyses. *SIGPLAN Not.* **47**(6), 217–228 (2012), ISSN 0362-1340, <https://doi.org/10.1145/2345156.2254091>
- [3] Babic, D., Hu, A.J.: Calysto: scalable and precise extended static checking. In: *Proceedings of the 30th International Conference on Software Engineering*, p. 211–220, ICSE ’08, Association for Computing Machinery, New York, NY, USA (2008), ISBN 9781605580791, <https://doi.org/10.1145/1368088.1368118>
- [4] Beyer, D., Kanav, S., Kleinert, T., Richter, C.: Construction of verifier combinations from off-the-shelf components. *Form. Methods Syst. Des.* **66**(1), 99–130 (2025), ISSN 0925-9856, <https://doi.org/10.1007/s10703-024-00449-y>
- [5] Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: *Proc. TACAS* (3), pp. 151–186, LNCS 15698, Springer (2025), https://doi.org/10.1007/978-3-031-90660-2_9
- [6] Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Bjørner, D., Broy, M., Pottosin, I.V. (eds.) *Formal Methods in Programming and Their Applications*, pp. 128–141, Springer Berlin Heidelberg, Berlin, Heidelberg (1993), ISBN 978-3-540-48056-3
- [7] Calcagno, C., Distefano, D., O’Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. *J. ACM* **58**(6) (2011), ISSN 0004-5411, <https://doi.org/10.1145/2049697.2049700>
- [8] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, California, USA, January 1977, pp. 238–252, ACM (1977), <https://doi.org/10.1145/512950.512973>
- [9] Cousot, P., Cousot, R.: Abstract interpretation frameworks. *J. Log. Comput.* **2**(4), 511–547 (1992), <https://doi.org/10.1093/LOGCOM/2.4.511>
- [10] Dewey, K., Kashyap, V., Hardekopf, B.: A parallel abstract interpreter for javascript. In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, p. 34–45, CGO ’15, IEEE Computer Society, USA (2015), ISBN 9781479981618
- [11] Grahn, H.: Transactional memory. *J. Parallel Distributed Comput.* **70**(10), 993–1008 (2010), <https://doi.org/10.1016/J.JPDC.2010.06.006>
- [12] Helm, D., Kübler, F., Kölzer, J.T., Haller, P., Eichberg, M., Salvaneschi, G., Mezini, M.: A programming model for semi-implicit parallelization of

- static analyses. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, p. 428–439, ISSTA 2020, Association for Computing Machinery, New York, NY, USA (2020), ISBN 9781450380089, <https://doi.org/10.1145/3395363.3397367>
- [13] Hong, J., Ryu, S.: Concrat: An Automatic C-to-Rust Lock API Translator for Concurrent Programs. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 716–728 (2023), ISSN 1558-1225, <https://doi.org/10.1109/ICSE48619.2023.00069>
 - [14] Jordan, H., Subotić, P., Zhao, D., Scholz, B.: Specializing parallel data structures for datalog. Concurrency and Computation: Practice and Experience **34**(2), e5643 (2022), URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5643>, e5643 cpe.5643
 - [15] Kim, S.K., Venet, A.J., Thakur, A.V.: Deterministic parallel fixpoint computation. Proc. ACM Program. Lang. **4**(POPL) (2019), <https://doi.org/10.1145/3371082>
 - [16] Kocal, A.R., Schwarz, M., Saan, S., Seidl, H.: Same engine, multiple gears: Parallelizing fixpoint iteration at different granularities (2025), <https://doi.org/10.5281/zenodo.18186778>, URL <https://doi.org/10.5281/zenodo.18186778>
 - [17] Kocal, A.R., Schwarz, M., Saan, S., Seidl, H.: Same engine, multiple gears: Parallelizing fixpoint iteration at different granularities (extended version) (2026), URL <https://arxiv.org/abs/2602.06680>
 - [18] Lemberger, T., Wachowitz, H.: Nacpa: Native checking with parallel portfolio and analyses: (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems: 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3–8, 2025, Proceedings, Part III, p. 236–241, Springer-Verlag, Berlin, Heidelberg (2025), ISBN 978-3-031-90659-6, https://doi.org/10.1007/978-3-031-90660-2_18
 - [19] McPeak, S., Gros, C.H., Ramanathan, M.K.: Scalable and incremental software bug detection. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, p. 554–564, ESEC/FSE 2013, Association for Computing Machinery, New York, NY, USA (2013), ISBN 9781450322379, <https://doi.org/10.1145/2491411.2501854>
 - [20] Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Burns, J.E., Moses, Y. (eds.) Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23–26, 1996, pp. 267–275, ACM (1996), <https://doi.org/10.1145/248052.248106>
 - [21] Monniaux, D.: The parallel implementation of the astrée static analyzer. In: Yi, K. (ed.) Programming Languages and Systems, pp. 86–96, Springer Berlin Heidelberg, Berlin, Heidelberg (2005), ISBN 978-3-540-32247-4
 - [22] Monniaux, D.: The parallel implementation of the astrée static analyzer. In: Yi, K. (ed.) Programming Languages and Systems, pp. 86–96, Springer Berlin Heidelberg, Berlin, Heidelberg (2005), ISBN 978-3-540-32247-4

- [23] Oki, B., Pfluegl, M., Siegel, A., Skeen, D.: The information bus: an architecture for extensible distributed systems. *SIGOPS Oper. Syst. Rev.* **27**(5), 58–68 (1993), ISSN 0163-5980, <https://doi.org/10.1145/173668.168624>
- [24] Peringer, P., Šoková, V., Vojnar, T.: Predatorhp revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: Biere, A., Parker, D. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 408–412, Springer International Publishing, Cham (2020), ISBN 978-3-030-45237-7
- [25] Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: Cytron, R.K., Lee, P. (eds.) *Conference Record of POPL’95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, USA, January 23–25, 1995, pp. 49–61, ACM Press (1995), <https://doi.org/10.1145/199448.199462>
- [26] Rodríguez, J., Lhoták, O.: Actor-based parallel dataflow analysis. In: Knoop, J. (ed.) *Compiler Construction*, pp. 179–197, Springer Berlin Heidelberg, Berlin, Heidelberg (2011), ISBN 978-3-642-19861-8
- [27] Sahebolamri, A., Gilray, T., Micinski, K.: Seamless deductive inference via macros. In: *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, p. 77–88, CC 2022, Association for Computing Machinery, New York, NY, USA (2022), ISBN 9781450391832, <https://doi.org/10.1145/3497776.3517779>
- [28] Scholz, B., Jordan, H., Subotić, P., Westmann, T.: On fast large-scale program analysis in datalog. In: *Proceedings of the 25th International Conference on Compiler Construction*, p. 196–206, CC ’16, Association for Computing Machinery, New York, NY, USA (2016), ISBN 9781450342414, <https://doi.org/10.1145/2892208.2892226>
- [29] Schwarz, M.: Thread-Modular Abstract Interpretation: The Local Perspective. Ph.D. thesis, Technical University of Munich (2025), URL <https://d-nb.info/137113345X/34>
- [30] Schwarz, M., Saan, S., Seidl, H., Erhard, J., Vojdani, V.: Clustered relational thread-modular abstract interpretation with local traces. In: *European Symposium on Programming*, pp. 28–58, Springer (2023), https://doi.org/10.1007/978-3-031-30044-8_2
- [31] Seidl, H., Vogler, R.: Three improvements to the top-down solver. *Math. Struct. Comput. Sci.* **31**(9), 1090–1134 (2021), <https://doi.org/10.1017/S0960129521000499>
- [32] Sherman, E., Dwyer, M.B.: Structurally defined conditional data-flow static analysis. In: Beyer, D., Huisman, M. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 249–265, Springer International Publishing, Cham (2018), ISBN 978-3-319-89963-3
- [33] Shi, Q., Xiao, X., Wu, R., Zhou, J., Fan, G., Zhang, C.: Pinpoint: fast and precise sparse value flow analysis for million lines of code. *SIGPLAN Not.* **53**(4), 693–706 (2018), ISSN 0362-1340, <https://doi.org/10.1145/3296979.3192418>
- [34] Shi, Q., Zhang, C.: Pipelining bottom-up data flow analysis. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*,

- p. 835–847, ICSE '20, Association for Computing Machinery, New York, NY, USA (2020), ISBN 9781450371216, <https://doi.org/10.1145/3377811.3380425>
- [35] Sivaramakrishnan, K.C., Dolan, S., White, L., Jaffer, S., Kelly, T., Sahoo, A., Parimala, S., Dhiman, A., Madhavapeddy, A.: Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.* **4**(ICFP), 113:1–113:30 (2020), <https://doi.org/10.1145/3408995>
 - [36] Stade, Y., Tilscher, S., Seidl, H.: Partial correctness of the top-down solver. *Archive of Formal Proofs* (2024)
 - [37] Stade, Y., Tilscher, S., Seidl, H.: The top-down solver verified: Building confidence in static analyzers. In: Gurfinkel, A., Ganesh, V. (eds.) *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I, Lecture Notes in Computer Science*, vol. 14681, pp. 303–324, Springer (2024), https://doi.org/10.1007/978-3-031-65627-9_15, URL https://doi.org/10.1007/978-3-031-65627-9_15
 - [38] Stemmler, F., Schwarz, M., Erhard, J., Tilscher, S., Seidl, H.: Taking out the toxic trash: Recovering precision in mixed flow-sensitive static analyses. *Proc. ACM Program. Lang.* **9**(PLDI) (2025)
 - [39] Stiévenart, Q., Van Es, N., Van der Plas, J., De Roover, C.: A parallel worklist algorithm and its exploration heuristics for static modular analyses. *J. Syst. Softw.* **181**(C) (2021), ISSN 0164-1212, <https://doi.org/10.1016/j.jss.2021.111042>
 - [40] Sun, Z., Xu, D., Zhang, Y., Qi, Y., Wang, Y., Zuo, Z., Wang, Z., Li, Y., Li, X., Lu, Q., Peng, W., Guo, S.: Bigdataflow: A distributed interprocedural dataflow analysis framework. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, p. 1431–1443, ESEC/FSE 2023, Association for Computing Machinery, New York, NY, USA (2023), ISBN 9798400703270, <https://doi.org/10.1145/3611643.3616348>
 - [41] Van Es, N., Stiévenart, Q., Van der Plas, J., De Roover, C.: A parallel worklist algorithm for modular analyses. In: *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 1–12 (2020), <https://doi.org/10.1109/SCAM51674.2020.00006>
 - [42] de Vilhena, P.E., Pottier, F., Jourdan, J.: Spy game: verifying a local generic solver in iris. *Proc. ACM Program. Lang.* **4**(POPL), 33:1–33:28 (2020), <https://doi.org/10.1145/3371101>
 - [43] Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: the goblin approach. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pp. 391–402, ACM (2016), <https://doi.org/10.1145/2970276.2970337>
 - [44] Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 351–363, POPL '05, Association for

Computing Machinery, New York, NY, USA (2005), ISBN 158113830X,
<https://doi.org/10.1145/1040305.1040334>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Gillian Debugging: Swinging Through the (Compositional Symbolic Execution) Trees

Nat Karmios , Sacha-Élie Ayoun , and Philippa Gardner 

Imperial College London, UK
n.karmios@ic.ac.uk

Abstract. In recent years, compositional symbolic execution (CSE) tools have been growing in prominence and are becoming more and more applicable to real-world codebases. Still to this day, however, debugging the output of these tools remains difficult, even for specialist users. To address this, we introduce a debugging interface for symbolic execution tools, integrated with Visual Studio Code and the Gillian multi-language CSE platform, with strong focus on visualisation, interactivity, and intuitive representation of symbolic execution trees. We take care in making this interface tool-agnostic, easing its transfer to other symbolic analysis tools in future. We empirically evaluate our work with a user study, the results of which show the debugger’s usefulness in helping early researchers understand the principles of CSE and verify fundamental data structure algorithms in Gillian.

Keywords: Debugging, symbolic execution, verification

1 Introduction

Symbolic execution (SE) is a static program analysis technique that explores program behaviour by executing code over symbolic (rather than concrete) inputs. Although symbolic execution tools and frameworks based on first-order logic, such as CBMC [5] and KLEE [4], are widely used in academia and industry, their reasoning does not scale when faced with heap-manipulating programs.

Compositional symbolic execution [14, 15] (CSE) addresses this limitation by extending SE with function specifications expressed in a separation logic, such as the original separation logic (SL) [22] and the recent incorrectness separation logic (ISL)[24]. This enables functions to be analysed in isolation, independent of their calling contexts, and records the results as concise specifications for reuse during subsequent analyses. This compositionality significantly improves scalability while retaining precise reasoning about heap-manipulating programs.

The development and adoption of CSE-based tools has steadily increased both in academia and industry. For example, VeriFast provides semi-automatic compositional verification for C, Java and Rust using SL; C/Ns supports verification of C programs using SL; the Viper [21] framework, based on implicit dynamic frames (akin to SL), underpins verification tools for Go, Java, Python

and Rust [2, 6, 26]; Infer-Pulse [25], built on ISL, is deployed at Meta for automated multi-language bug finding at scale [17]; and Gillian [3, 8, 16] offers a unified framework for correctness and incorrectness reasoning based on SL and ISL respectively, applied to C, JavaScript and Rust. These tools have reached a level of maturity that enables their application to real-world software, including the AWS SDK (Gillian), Google’s pKVM hypervisor (CN), internet routers (Viper), Meta’s production codebase (Infer-Pulse), and the Rust standard library (Gillian, VeriFast). Several are also used in teaching program analysis principles to undergraduate and graduate students (Gillian, VeriFast, Viper).

With this growing and multi-faceted applicability comes a need for clearly presenting and efficiently debugging the output of a given CSE tool. To us, this need is threefold: (1) to allow tool developers to examine the inner workings of the tool in full detail so that they can improve its implementation; (2) to enable tool users to debug their programs by relating the output of the tool to the analysed code; and (3) to guide students, who are the next generation of tool users and developers, in understanding the fundamentals of CSE reasoning. This, however, still remains a relatively uninvestigated topic, typically addressed in an ad-hoc fashion, with user experience varying greatly from tool to tool: some only provide (overwhelming) textual feedback; some offer minimal language-server-based integration; others offer interactive exploration but limit it to generated counter-examples or are post-hoc; and the few that do have truly interactive debuggers either adopt broad editor integration at the cost of visual expression or tightly focus on integrating with a single editor, making the transfer to other tools difficult. A more detailed overview of the state-of-the-art is given in §2.

To address this gap, we introduce an interactive debugger for CSE-based verification, using Gillian as a case study. Our goal is to design a tool-agnostic debugger, identifying which of its components are general enough to support other CSE tools, and which are specific to Gillian. Gillian serves as an ideal test bed for this work due to its parametric treatment of memory, support for multiple source languages through compilation to a common intermediate representation (IR), and implementation of multiple analyses within a unified framework (§3). Our debugger is integrated with Visual Studio Code [18] to leverage a widely used and familiar development environment, again taking a generalised approach intended to facilitate future integration with additional editors beyond VSCode.

We give a tour of the debugging experience in §4 and present the principles underlying our debugger in §5, focusing on:

- *execution tree representation*, where we aim to create a useful, intuitive, and flexible representation of analysis with a tree-of-trees structure;
- *execution tree capture*, using this tree-of-trees structure to record information about the analysis in an unobtrusive way at the engine level;
- *execution tree lifting*, which lifts a low-level tree-of-trees structure to a source-level one, which can be read by the user; and
- *debugging interactivity*, giving the user full control of examining and directing the symbolic analysis on-the-fly rather than summarising it after completion.

Importantly, we take care to decouple our design choices from the Gillian implementation whenever possible, making them transferable to other CSE tools, and even potentially more traditional symbolic execution tools. Further, in §6, we highlight interesting lessons and observations that have come up during the substantial engineering work involved in this project.

We performed a basic user study at a PL summer school (§7), evaluating the usability of the Gillian debugger on CSE-base verification. Early researchers were introduced to SL, CSE and Gillian, before using Gillian to try to verify a number of data-structure algorithms. The examples were designed to require use of the main features of semi-automatic compositional verification: specification of pre- and post-conditions, loop invariants, and proof tactics. Afterwards, we performed a quantitative analysis that revealed that the users spent a substantial amount of time working with the debugger, indicating its usefulness. We also collected qualitative feedback using a Lickert-style questionnaire, with the aggregated results indicating that working with the debugger was helpful and intuitive, and its feedback informative and educational.

2 Related Work

We briefly review the debugging capabilities of state-of-the-art CSE tools. Gillian (prior to our work) and CN provide only text-based logs, which in real-world verification can reach tens of millions of lines. These logs linearise the inherently non-linear analysis into flat textual streams that, while exhaustive, are overly detailed and interpretable only by experts familiar with the tools.

Infer-Pulse generates HTML files that present similar information to text logs, organised into analysis “nodes” associated with source code locations. Although this offers greater clarity than plain logs, it remains difficult to follow the flow of the analysis through the program, and the need to leave the development environment disrupts the user’s workflow.

Viper integrates with editors via the Language Server Protocol (LSP) [20], re-running analyses as the user types and reporting failures inline. This proves useful for receiving rapid feedback, though such feedback is often limited to brief error messages. We argue that this approach is complemented by the availability of a debugger when deeper insight into the analysis is needed.

Finally, VeriFast, through its VSCode extension [11], enables interactive inspection of the analysis’ execution tree, akin to a debugging session. However, the tree nodes are unlabelled and difficult to relate back to the source code. Moreover, as analysis terminates at the first breakpoint or failure, the post-hoc tree is typically incomplete, leaving some execution branches unexplored.

Our Gillian debugger extends the Debug Adapter Protocol (DAP) [19] to provide an interactive symbolic debugging environment for its CSE analyses. Several other symbolic analysis tools, such as GobPie [10] and SecC [7], also use DAP to deliver an integrated debugging experience with many editors “for free”. However, the DAP offers no intuitive mechanism for representing branching execution, leading ad-hoc solutions such as repurposing the DAP’s ‘threads’ view, originally designed for visualising concurrent threads in concrete execution.

The debugger perhaps most similar to ours is the feature-rich Symbolic Execution Debugger [9] of the KeY project [1]. This debugger is capable of displaying detailed, annotated symbolic execution trees, but relies on a non-standard interface and is tightly integrated with both the Eclipse IDE and KeY’s first-order Java analyses, limiting its portability to other IDEs and analysis frameworks.

3 Background

Symbolic Execution is a program analysis technique that executes a given program with symbolic instead of concrete inputs, where one symbolic state describes a set of concrete states, enabling general reasoning about program behaviour. Symbolic execution can branch (e.g. via an if-else statement), with each branch carrying a logical assertion called a *path condition* which describes the constraints on symbolic variables that have led the execution to that branch.

Compositional Symbolic Execution (CSE) extends symbolic execution to use and create function specifications written in a separation logic. In particular, the reasoning allows function behaviour to be summarised on *partial* symbolic states using function specifications, which are Hoare triples that have meaning given by an underlying separation logic. With verification, the pre-condition describes a part of the symbolic state that is sufficient for the function to be evaluated; the post-condition describes all the possible results.

When calling a function, a specification of the callee can be used instead of inlining its body, providing compositionality and improving scalability. Executing a specification means *matching* the current symbolic state against its pre-condition, *consuming* (or ‘exhaling’) the matched part from the state, and *producing* (or ‘inhaling’) the post-condition. Verifying a function simply requires symbolically executing the function from its pre-condition until termination, then matching each final state against the post-condition. Since function calls are replaced by the execution of their specifications, a symbolic execution tree is as long as the number of statements in a single function, drastically reducing reasoning complexity compared to non-compositional SE.

Gillian is a multi-language platform that supports three types of analysis: whole-program symbolic testing, providing bounded verification guarantees similarly to the first-order CBMC tool [5]; semi-automatic compositional verification à la CN, VeriFast using SL, and Viper; and automatic bug-finding similar to Infer-Pulse using ISL. Gillian is parametric on the memory model of the source language analysed, which needs to be provided per instantiation together with a compiler from the source language to Gillian’s intermediate representation, GIL. When it comes to real-world languages, Gillian has been instantiated to C, JavaScript, and Rust. For ease-of-presentation, this paper focuses on **WISL**, a demonstrator Gillian instantiation for a simple while language with a C-like block-offset memory model routinely taught to fourth-year undergraduates and MSc students. More detail on this language can be found in the extended version of this paper [12].

4 The Gillian Debugging Experience

We illustrate the features of the Gillian debugger by using it to debug the verification of the WISL `llen()` function (Fig. 1, left), which computes the length of a null-terminated singly-linked list (SLL), each node of which consisting of a two-cell block containing the value carried by the node and the pointer to the next node. The function is specified using the standard separation-logic SLL predicate, `list(x, alpha)`, which states that the SLL starting from block with identifier `x` contains the values in the mathematical list `alpha`. Then, in the pre-condition we have just the list, and in the post-condition we additionally state that the return value equals its length. To illustrate the difference between the source- (WISL) and the IR-level (GIL), we also give a stylised compilation of `llen()` to GIL in Fig. 1 (right), where we can see how WISL's structured control flow becomes unstructured via GIL `gotos`, and also how WISL commands (such as pointer dereferencing, `t := [x + 1]`) get broken down into several GIL steps.

```

1 predicate list(x, alpha) {
2   (x == null) * (alpha == nil);
3   (x -> #v, #z) * list(#z, #beta)
4   * (alpha == #v::#beta)
5 }
6
7 { (x == #x) * list(#x, #alpha) }
8 function llen(x) {
9   if (x == null) {
10    n := 0
11   } else {
12    t := [x+1]; n := llen(t)
13   };
14   return n
15 }
16 { list(#x, #alpha) * (ret == len(#alpha)) }

1 proc llen(x) {
2   goto? (x == null) then else;
3   then: n := 0;
4   goto end;
5   else: _var0 := i_add(x, 1);
6   goto? (_var0 is Ptr) cont fail;
7   fail: fail "Invalid pointer";
8   cont: t := load<_var0>;
9   n := llen(t);
10  end: skip;
11  ret := n;
12  return
13 };

```

Fig. 1. Slightly simplified SLL predicate and a specified WISL recursive list-length function (left), and the corresponding compiled GIL code, simplified (right). The prefix `#` denotes a logical variable.

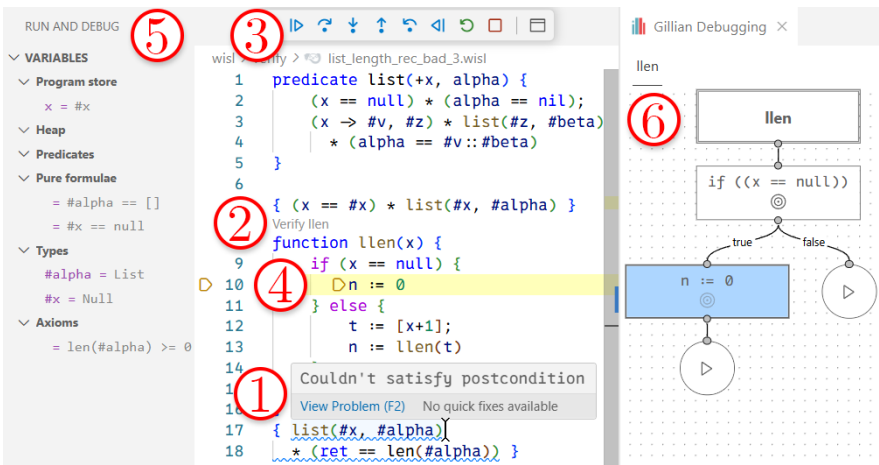


Fig. 2. Gillian's debugging and language server interface when verifying `llen()`.

When trying to verify `llen()` using Gillian and VSCode (cf. Fig. 2), the language server informs us that there is an error (①), stating that the function post-condition could not be satisfied. While other existing LSPs, such as Viper’s, may provide slightly more precise information, such as which subset of the post-condition failed, the LSP interface can only point to the failure, not explain it. At this point, we turn to the Gillian debugger to trace the root cause of the error, as the verification could have failed for a variety of reasons, such as an incorrect specification, a missing proof annotation, or a bug in the program itself.

Above each function, the Gillian debugger displays a ‘Verify’ button (②) that initiates an interactive debugging session for that function. While in a session, several pieces of interface are presented to the user: controls for stepping through the executions (③); a highlight of the line of code being currently executed (④); a breakdown of the current symbolic state (⑤); and a tree view that visualises the symbolic execution presented in a separate panel (⑥).

The controls at ③ are inherited from the VSCode DAP interface. They contain: a ‘continue’ button, which continues execution along the current branch until the next breakpoint or end of execution; a ‘step over’ button which executes the next WISL statement; a ‘step in’ button which steps inside a function call if present; a ‘step out’ button which continues execution until the current function returns; a ‘step back’ button which undoes the last statement; a ‘continue backwards’ button which returns to the previous breakpoint or start of execution; a ‘restart’ button which restarts the entire symbolic execution from the beginning; and a ‘stop’ button, which ends the debugging session.

When navigating through the symbolic execution using these controls, the line of code currently being executed is highlighted with a small arrow pointing to the sub-expression being evaluated (④). In addition, the current symbolic state is displayed on the left-hand side of the screen, using VSCode’s built-in variable explorer to display the current program variable store, heap, predicates, path conditions, and other relevant information (⑤).

Finally, and most importantly, the tree view (⑥) enables interactive visualisation of the symbolic execution. Statements are represented as nodes in a tree structure, with edges representing the flow of execution. Clicking on a node jumps to that point in the execution, updating the highlighted code and symbolic state, providing a way to navigate *between* branches as well as forward and backward within the same branch. If a branch has not yet terminated, a ‘play’ button is displayed as a leaf of that branch and can be clicked on to execute the next statement on that branch. When examples increase in complexity and verification times grow longer, this interface allows users to contain branch exploration, avoiding long waiting times.

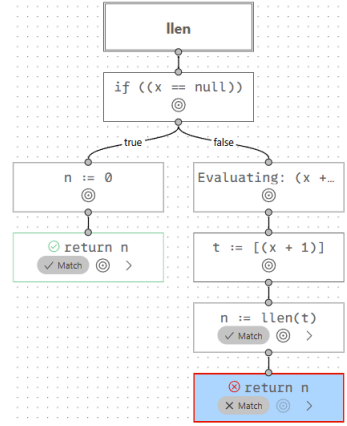


Fig. 3. Full symbolic execution tree of `llen()`

Fig. 3 shows the full symbolic execution tree of `llen()`, from which we identify that the path leading to a verification failure is the one taking the ‘false’ branch of the `if` statement, as identified by the red cross next to the return statement. This node also displays a small arrow pointing to the right, indicating that there is *nested* information within that node.

Clicking that arrow expands the node to reveal the steps of the post-condition matching that failed (Fig. 4), and the magnifying glass icon in the nested tree ‘pops it out’ to a new tab to view in isolation. Notice how the visual branching remains relevant in matches, where Gillian has backtracked and attempted to recover by unfolding the list predicate. Within this tree, we see that Gillian failed to match `ret == len(#alpha)` (leftmost branch) before backtracking and applying a *recovery tactic* (here, the ‘unfold’ tactic), where the match still fails for the same reason (two rightmost branches). Recovery tactics are triggered by heuristics and sometimes lead to confusion; this example illustrates how our debugger clearly displays this process to the user.

By clicking on the leftmost failing node and then inspecting the symbolic state in detail (Fig. 4, right), we see that `ret` equals `len(#lvar_4)`, but `#alpha` is the list `([#lvar_7] @ #lvar_4)`, and that therefore `ret` is off by one. This is the final step that leads to the understanding of the root cause of the error: we forgot to increment `n` in the program itself. In particular, after line 12 of Fig. 1 (left), `n := n + 1` is required. While this last step must still be taken by the user, the information required to get them there was accessible within a few clicks.

We note that, at the top level, the maximum depth of the execution tree is the number of statements in the function being verified (modulo expression evaluation, which is an implementation choice); as CSE permits verifying functions in isolation, execution tree sizes are constrained. While real-world conditions—e.g. larger functions, more complex source languages—will naturally produce trees larger than we encounter in this work, we expect them to remain orders of magnitude smaller than those seen in non-compositional symbolic analyses à la KLEE.

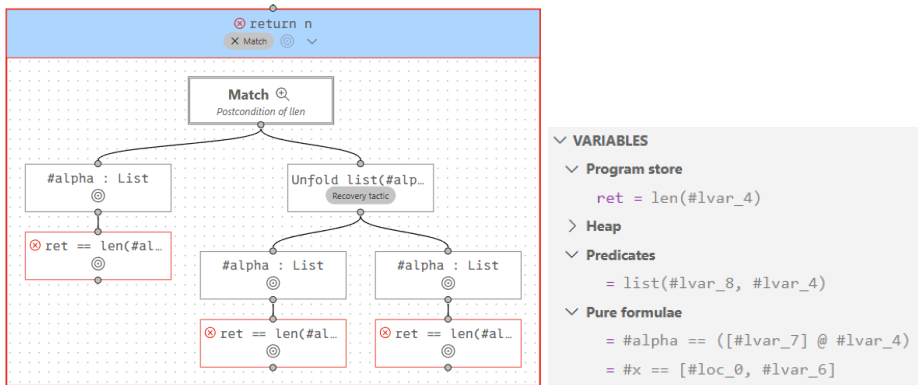


Fig. 4. Expanded failing node (left) and symbolic state at the failing match (right).

5 Principles of CSE Debugging

We give an overview of our three-layered architecture in Fig. 5 and discuss the key processes we compose to create a usable CSE debugger. We will refer to the execution of the WISL statements $t := [x+1]; n := \text{llen}(t)$ on line 12 of `llen()`, which correspond to lines 5–8 and 9 of the GIL code respectively (see Fig. 1). We abstract these concepts from Gillian-specific implementation details of Gillian to encourage wider applicability; some implementation details specific to Gillian are discussed in §6.

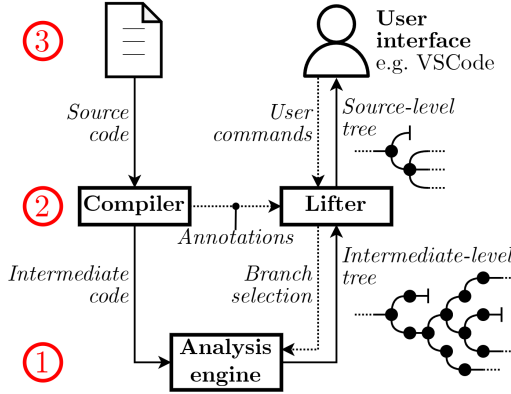


Fig. 5. Overview: architecture of the Gillian debugger.

At Layer 1, we create a tree structure that describes the analysis process at the level of the intermediate representation (IR). This is a low-level tree that includes all the execution steps of the analysis, and, ideally, any associated fundamental processes (for Gillian, for example, these would be matching (supported) and expression and state simplifications (currently unsupported)). This tree must accurately represent one’s intuition of the ‘shape’ of the analysis, and be extensible over time in the context of an interactive analysis. Working with this layer alone is only for the expert tool developer, giving them the ability to fine-tune tool performance and understand unexpected behaviours or bugs in the tool itself.

To present an understandable and actionable execution tree to a tool user working with the analysed source language, one must also bridge the disconnect between the internal IR of the tool and said source language. This is the purpose of Layer 2, at which the debugger employs various lifting mechanisms to abstract the low-level tree from Layer 1 into a source-level execution tree.

Finally, Layer 3 concerns an important but rarely discussed aspect of tool development: the infrastructure through which a user interacts with the tool. UI development, especially when aiming for reuse, maintainability, and longevity, can be an arduous engineering task, as it requires a deep understanding of existing developer tooling.

5.1 Low-level Tree-of-trees Structure

An intuitive, interactive debugging experience requires a structure that accurately represents the ‘shape’ of the analysis process, without knowing the execution order ahead of time. One could argue that symbolic execution can be represented simply as a tree, where a node with multiple children denotes symbolic branching. We represent analyses at both the intermediate and source level with a *tree-of-trees* structure, where each node can contain nested trees providing further detail. This has the benefit of capturing the intuition of breaking execution and analysis down into smaller steps, while hiding excessive details until desired by the user. Nesting is used, for example, to represent Gillian’s matching process or the body of an inlined function call; recall `llen()`’s execution tree given in Fig. 3, where the failing `return` node expanded to reveal the nested matching tree shown in Fig. 4.

A CSE tool will typically implement a particular execution strategy (e.g. depth-first) for traversing the execution tree. Interactive debugging requires recording an execution tree that is agnostic to any such strategy, in such a way that it can handle interrupting and later returning to an execution path. The tools’ analysis engine will therefore need some modification to provide this information necessary to build the tree-of-trees. However, tracking non-trivial control flow in symbolic execution, such as entering and leaving function and loop bodies, can dramatically increase engine complexity, making a critical part of the tool more difficult to maintain. We found that a minimal approach is sufficient, and can straightforwardly apply to many CSE tools: instead of just writing text logs describing the state of analysis at each step, like most tools do, the engine can produce a machine-readable report with similar information and a unique identifier. A report also stores the identifier of the previous execution step, and the enclosing step that nests it (if relevant); this requires only light bookkeeping from the engine and can easily be stored alongside the execution context. These reports can be freely accessed later in the debugging pipeline, where the heavy lifting of constructing a more intricate tree can be performed with no further input from the engine. Fig. 6 shows a GIL trace segment for the WISL statements on line 12; to keep engine logic simple, function bodies such as that of `i_add()` are not nested under the calling command. Nesting is still utilised at the intermediate level where convenient, such as matching for the recursive call to `llen()`.

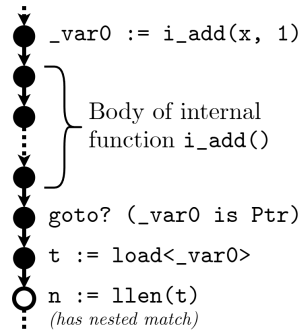


Fig. 6. A snippet of `llen()`’s GIL trace.

5.2 Lifting to the Source Level

After applying the prior concepts, one is rewarded with a perfectly adequate interactive execution tree, but only for the relevant tool’s IR. To facilitate debugging at the level of the source language, a CSE debugger needs to include a

lifter, whose responsibility is to tell the analysis engine how to transform the tree-of-trees for a specific IR into a tree-of-trees more reminiscent of a particular source language—in our debugger, from GIL to WISL. A lifter can, for example, transform the shape of a tree by combining the nodes of multiple IR-level commands into a single node for the originating source-level statement, it could alter and combine branch cases for greater clarity, and it could introduce nesting where appropriate. A lifter can also improve the information associated with individual nodes by, for example, producing node display text and transforming the displayed state to be more recognisable at the source level.

The source-aware information needed to construct a source-level tree-of-trees is most easily gathered during compilation and can be passed to the lifter via **annotations** on IR code. These annotations may include the display text for the source statement, flags denoting which groups of IR statements comprise a source statement, or hints at the reason a branch may occur. As an example, the WISL statement `t := [x+1]` compiles to lines 5–8 of the GIL code: we choose to count expression evaluation as a distinct step, so line 5 is flagged as terminating a node; line 6 is flagged as *not* terminating a node; and lines 7 and 8 are flagged as terminating. In the source-level tree, this results in one step for evaluating `x+1`, and one for performing the lookup. See the extended paper for some further details on WISL’s annotations and their assignments to `llen()`’s GIL code.

5.3 User Interface

When it comes to UI development, given the amount of engineering work it requires, one should try to avoid ‘reinventing the wheel’ to create an interface with a slightly different functionality or in a slightly different environment.

As discussed in §2, some tools leverage the DAP to take advantage of debugging interfaces provided by IDEs such as VSCode. This includes, for example, highlighting the current code step, controls for stepping through execution, and a view of the current state. The DAP, however, is not expressive enough to represent branching execution, much less navigate a complete execution tree, leading to ad-hoc solutions such as using the ‘threads’ view to list execution branches, or abandoning the DAP altogether in favour of a bespoke post-analysis tree viewer.

Our approach achieves the best of both worlds by *extending* the DAP to support interaction with a custom tree view in the editor. Referring to Fig. 2, the standard debugging UI elements are provided as standard for a DAP-implementing debugger; users can see the line of code (④) and the state (⑤) at the current step, and use the (linear) stepping controls (③). The bespoke tree view (⑥) is supported by a custom event `mapUpdate` to relay changes to the tree, and the custom commands `jump` to travel to a different point in the tree and `stepSpecific` to step forward via a specific branch when multiple are available. This does detract from the DAP’s editor agnosticism—due to requiring a VSCode extension—but we believe this to be a worthwhile compromise for the moment, and are working on re-establishing DAP editor independence.

6 Implementation Insights

While the specifics Gillian Debugging’s implementation are not a primary contribution of this work, we would like to highlight a number of observations that have emerged and that could be of interest to the wider community.

Structured logs. The structured logs required by the debugger are necessarily more complex to keep track of than regular logs. After experimentation, we decided that the burden of maintaining the structure should be minimised for the symbolic execution engine, an already complex piece of technology with strong correctness requirements. Specifically, we separate concerns: the engine performs minimal bookkeeping, storing all log reports in an SQLite database with only two additional fields, `previous` and `parent`; the lifter later uses these reports to construct a tree with more elaborate control flow. The former operation is language-independent and is factored out from Gillian instantiations.

Rebuilding source-level commands. The WISL lifter aggregates executed GIL steps into WISL commands. Each executed GIL step is assigned to a *partial command* representing a source-level WISL command that has not yet been fully explored. Annotations are used to mark GIL commands that terminate a WISL command, allowing the lifter to finalise a partial command into a source-level tree node when all paths have been explored. As we had full control over the WISL compiler, we could straightforwardly extend it to add such annotations to the GIL code. This approach is applicable to other compilers: for example, we successfully applied the same pattern to a new Gillian-C instantiation that uses CBMC’s parser and IR. We could not adapt the original Gillian-C instantiation, as the CompCert compiler it employs is written in Rocq and is therefore difficult to modify. This demonstrates that our approach can be portable across different language implementations if the frontend preserves sufficient source-level information during compilation.

Advanced nesting. The tree-of-trees structure and its visual nesting (cf. Fig. 4) were particularly useful in reifying source-level nesting where the IR-level tree had none. For instance, WISL loops annotated with invariants compile to GIL functions with pre- and post-conditions, but the WISL lifter is able to re-integrate the loop body by initiating its separate verification and nesting the resulting tree in the tree of the outer function. This demonstrates both the flexibility of the tree-of-trees structure and the ability of the lifter to reconstruct source-level control flow that fundamentally changes during compilation.

Everything is interactive. A key difference between our approach and post-analysis tree viewers is the ability for users to interactively guide the analysis as it progresses. To implement this interactivity, we modified Gillian’s interpreter to return a continuation function after each GIL execution step. In normal execution mode, these continuations are simply called repeatedly until completion, preserving the original behaviour. In debugging mode, however, the continuations are called with a branch identifier, selecting which execution path to

continue. This approach shows promise for integration with other symbolic execution tools that already operate in continuation-passing style, such as VeriFast and Viper, or with continuation monads, such as CN.

The lifter cooperates with this engine-level interactivity by translating between user-level commands and engine operations. For each high-level instruction (‘step in’, ‘step over’, etc.), the lifter determines the appropriate GIL steps required to extend the tree.

A generalised UI toolkit. Gillian’s debugging UI is implemented in VSCode by combining extensions to the DAP with a dedicated tree-viewing interface presented via a web-view panel. Early iterations of the debugger revealed that both the extensions to the DAP and the custom tree viewing interface had few conceptual dependencies on Gillian or VSCode, and could be separated entirely; the interface deals in abstract steps, requiring no knowledge of Gillian, its IR, or the language being analysed, and said interface is contained in a single web-based UI component. This realisation has been brought to fruition with the working title of SEDAP (Symbolic Execution DAP), a prototype for a standard extension to the DAP with libraries providing the interactive tree view as an extensible web component, and helpers for integrating this interface with VSCode debug sessions. Just as the DAP bridges editors with traditional debuggers, further work on SEDAP should bridge editors with analysis tools like those discussed in §2; Gillian’s debugger in VSCode now serves as the first example of this in practice.

Benchmark	Time to verify (ms)		
	No logs	File	Database
SLL_iterative.wisl	242	680 (2.8x)	487 (2.0x)
DLL_recursive.wisl	91	259 (2.8x)	174 (1.9x)
sll.c	303	390 (12.8x)	668 (2.2x)
dll.c	1875	81501 (43.5x)	6335 (3.4x)
priQ.c	292	5730 (19.6x)	7815 (2.7x)
sort.c	236	2369 (10.0x)	475 (2.0x)
SLL.js	149	1192 (8.0x)	1188 (8.0x)
DLL.js	202	1624 (8.1x)	1672 (8.3x)
ExprEval.js	246	6415 (26.1x)	4513 (18.3x)

Table 1. Benchmark results on a selection of Gillian’s verification examples across WISL, Gillian-C and Gillian-JS. This compares no logging, verbose file logging, and structured database logging, showing slowdown factor for the latter two. Benchmarks were performed on a laptop with an Intel i7-1065G7 processor and 32 GiB of memory.

Performance impact. As shown in Tbl. 1, structured logging to the on-disk database incurs a performance cost over no logging, but is no worse than that of Gillian’s default text log. Interestingly, database logs fare much better against file logs in C verification, owing to the complicated logic used to print the heap; this demonstrates a benefit of separating logging concerns from the engine. We note that no effort has yet been made to optimise structured logs, and there is much

low-hanging fruit for performance improvements: database queries could be performed asynchronously or in batches; a different database model could be more appropriate (say, a graph database rather than relational); and a more efficient serialisation method such as protocol buffers could be used instead of JSON.

7 Empirical Evaluation

We made a formative attempt at evaluating Gillian debugging as part of a summer school attended by late-stage undergraduate and early-stage PhD students interested in formal methods and PL research. Following 3 hours of lectures on SL, CSE and Gillian, participants were invited to spend two 90-minute lab sessions during which they would attempt to verify a number of WISL programs using Gillian’s language server and debugger. For this lab, we created a suite of 13 introductory and 6 advanced exercises: the introductory ones use simple programs on linked lists to introduce various aspects of Gillian verification, whereas the advanced ones consist of larger programs using more complex data structures (e.g., sets, doubly-linked-lists, and binary search trees) as well as a WISL translation of a part of the Collections-C library [23]; see the extended paper for the full list of exercises.

The lab session was initially guided in that we solved the first few exercises together with the participants, introducing them to the debugger interface and the Gillian proof tactics that they had at their disposal: folding and unfolding user-defined predicates, the `assert` and `apply` proof tactics, and loop invariants.

At the end of proceedings, participants were invited to complete an anonymised survey about their experience and to optionally submit usage logs. These logs were generated and stored locally on the participants’ devices and tracked a small amount of timestamped information per participant interaction with the debugger and language server. Of the ≈ 30 lab participants, 19 completed the survey, of which 18 submitted usage logs. We use this data to evaluate the usefulness of Gillian debugging in understanding CSE and completing the provided exercises. We first evaluate qualitatively by analysing responses to a series of Lickert-scale statements about the lab to gauge subjective participant impressions on the usefulness of the debugger. We then contrast these responses against a quantitative metric by analysing the provided usage logs to understand the actual participant use of the debugger and language server.

The provided Lickert-style statements and their combined responses are shown in Tbl. 2. The overall sentiment towards the usefulness of the debugger was overwhelmingly positive (statements 1/5/6), with 63% of responses strongly in favour and 26% somewhat so. Further, the participants felt that the debugger was well-integrated with the verification process (statement 10) and that other tools could benefit from similar debuggers (statements 11/12). The participants also recognised the usefulness of more specific properties arising from the interactive, step-by-step nature of the debugger, including the exploration of individual execution paths (statement 7) and viewing the state at each execution step (statement 9). They were less positive, however, about the viability of the language server alone (statements 2/8), with neutral-to-slightly-negative responses.

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1 I didn't need to use the debugger to complete the exercises	11	6	1	1	0
2 The language server was sufficient for me to complete the exercises	1	6	5	6	1
3 The debugger helped me to understand CSE	0	2	5	5	7
4 The debugger helped me understand what Gillian is doing	0	0	2	8	9
5 I found the debugger helpful in completing the exercises	0	0	3	4	12
6 The debugger helped me to understand why verification failed	0	2	3	6	8
7 I found it useful to explore each path of execution individually using the debugger	0	0	1	5	13
8 I understood the errors that the language server presented me with	1	7	3	4	4
9 I found it useful to see the state at each point in execution using the debugger	0	0	3	5	11
10 It felt natural to use the debugger as part of the verification workflow	0	1	2	4	12
11 This debugging interface would be applicable to similar tools I have used	0	1	4	3	6
12 I would benefit from a debugger like this in similar tools I have used	0	0	1	4	9

Table 2. Summed Lickert scale responses of the user survey.

On the quantitative side, the usage logs consist of timestamped entries for each participant interaction with the debugger and the language server; language server entries show the changes to the exercise file and the analysis result (success, or the reason for a failure), and debugger entries denote whether the user started or stopped a session or made a step through the tree. From this, we calculate the amount of time each participant spent on each question (accounting for switching between exercises and long periods of inactivity), and how much of that time was spent actively using the debugger (interpreted as having interacted with the debugger within the last 30 seconds), to build a rough estimate of what proportion of time participants spent using the debugger. The distribution of these proportions, shown in Fig. 7, suggests that the majority of participants spent between 40% and 60% of their time having recently interacted with the debugger, giving a strong indication that the participants indeed made frequent and active use of the debugger.

While this user study provided solid evaluation for Gillian debugging, a number of limitations prevented deeper insight. Ultimately, the lab primarily aimed to be engaging and informative for the participants, meaning the controlled conditions and intense observation desirable of a more formal user study were not

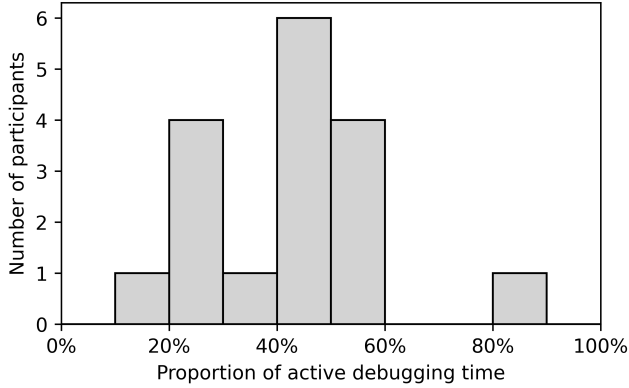


Fig. 7. Distribution: proportion of participant time in an active debugging session.

feasible. Participants were not required to focus entirely on the exercises at hand, nor work individually, nor stay for the whole runtime. Collecting more detailed information about participant behaviour would be difficult without more invasive techniques such as screen recording and diaries, which are arguably inappropriate in this context. Additionally, the students were still at an early stage in their PL-specific education; at least 60% of participants reported no prior experience with each of proof assistants, symbolic execution –based tools, CSE –based tools, and separation logic (see the extended paper). In contrast, the suite of exercises was created based on prior demonstrations of Gillian debugging to attendees of a fourth-year undergraduate course on separation logic. This resulted in less progress through the exercises than desired, with participants completing up to 6 of the 13 introductory exercises, and not reaching the advanced exercises.

After discussion with students and analysing these results, we believe that Gillian debugging has greater potential than simply assisting with verification tasks. We have formed new hypotheses on its use as a powerful tool for learning about CSE and its practical applications. For instance, we believe that newcomers to Gillian would use the debugger much more extensively than advanced users, who would reserve its use for more complex debugging tasks. While this hypothesis seems consistent with the positive responses to statements 3, 4 and 6, further study is required to assert it with greater confidence. Similarly, the participants’ inexperience with other CSE tools, together with the responses to statements 11 and 12, suggest that interactive visualisation could prove helpful for other analysis techniques such as theorem proving or abstract interpretation.

8 Lessons Learned & Future Work

Our experience developing and evaluating the symbolic execution debugger has revealed potential in making CSE tools more accessible, while highlighting work that needs to be done. We reflect on the lessons learned through this project, and outline directions for future research.

User study. Performing a user study is a rare occurrence in this space, and the efforts to perform one in this work were perhaps unexpectedly valuable; despite an evaluation of Gillian debugging being its primary motivation, it resulted in interesting discussions on the nature of tool usability and how it could vary for different analyses and levels of user experience. In addition to demonstrating educational value, future studies might target experienced tool users performing real-world verification tasks, observing in finer detail to perform focused evaluations of specific design choices. It may also be prudent to observe more objective metrics, such as the overall size or amount of branching in resulting trees.

Gillian. Due to its highly modular nature, Gillian proved to be a useful test bed for exploring debugging in this style, naturally leading to the separation of responsibilities between the compiler, analysis engine, and lifter. Its language-agnosticism provides the opportunity to explore the creation of symbolic execution trees for other languages without re-implementing the debugger internals, particularly to investigate debugging analyses of real-world programs using Gillian’s implementations for JavaScript and C. To this effect, a prototype lifter for our second Gillian-C has been created with the same techniques used for WISL. In addition to semi-automatic verification, Gillian can perform automatic, bounded, whole-program symbolic testing (with non-compositional symbolic execution) and true bug-finding with bi-abduction; further work will be able to explore the applicability of the debugging interface introduced in this work to these analyses.

Scaling analyses. Our focus on small, educational verification examples naturally raises the question of scaling to real-world programs, both in terms of lifter complexity for a real-world language and the interface’s ability to elegantly handle larger execution trees. Future work will apply our approach to analyses on non-trivial C programs, via both Gillian and other analysis tools such as CN.

Presenting the symbolic state. While this work primarily focuses on presenting an intuitive shape of analysis, more attention will be needed on the presentation of the symbolic state. For example, the WISL lifter prints the GIL-level expressions produced by the engine in a more WISL-esque syntax, but the state still includes intermediate variables introduced during compilation. These variables can be confusing to users as they seemingly have no basis in the source program, but are a necessary part of Gillian’s analysis on its intermediate representation. An investigation into how to meaningfully display CSE states at the source level could prove useful in improving user experience.

Error messages. Error messages are a notable pain point of this implementation, corroborated by the user survey (see Tbl. 2, statement 8). Verification error messages are difficult to implement, as it is often impossible to differentiate between an erroneous implementation, an erroneous specification, or insufficient proof annotations. Nonetheless, we do believe that Gillian’s error messages have room for improvement. Interactive debugging could turn an immutable error message into a conversation with the tool, exploring the entire analysis tree to refine the error into something more specific.

UI generality. Our DAP extensions and custom UI have already been decoupled from Gillian and VSCode to form the SEDAP. Further work will realise the goal of SEDAP to provide debugging for many symbolic analysis tools, connected to many well-known editors. Our hope is that this will in turn expand SEDAP’s capability to represent different internal processes, types of analyses, or volumes of information. How would SEDAP fare with abstract interpretation? Could it be extended to display symbolic heap visualisations? Could interactivity be leveraged to, for example, apply proof tactics in a live debugging session?

9 Conclusion

We have introduced a novel debugging interface for Gillian, addressing the underexplored challenge of creating intuitive debugging interfaces for compositional symbolic execution tools. We focus on providing a familiar user experience by leveraging VSCode’s existing debugging interface for standard UI elements, while developing custom components specifically designed to present compositional symbolic execution information in an intuitive way that accurately captures the analysis flow, including branching, nested information and state matching.

We have given the conceptual principles underlying our design, emphasising transferability of the approach to other tools, and discussed interesting technical insights that arose during implementation in the hope that they may inform future endeavours in this space. Importantly, we have decoupled the UI from Gillian and VSCode as much as possible, resulting in an independent library which we hope to continue developing and see adopted by other tools and editors.

Finally, we have conducted a preliminary evaluation of the debugger through a quantitative and qualitative analysis of its use by early researchers during a lab session, providing validation of its usefulness in understanding and working with compositional symbolic execution.

While the amount of engineering work involved to start proper research on improving the usability of compositional symbolic execution tools is significant—and several attempts have been made in the past and later abandoned—we hope that this work will serve as a stepping stone for future research in this space.

Data Availability Statement

An artifact containing Gillian binaries, the debugger VSCode extension, and example WISL programs (including the exercises used in evaluation) is available at [13].

Acknowledgements

We extend a heartfelt thanks to Opale Sjöstedt, Simon Park and Diego Cupello for their tireless work in creating exercises for the user study, as well as the participants of the study for making our evaluation possible; Jessica Shi and Alastair Donaldson for their advice on constructing the user study and this paper respectively; Petar Maksimović for his support in this project’s inception; and Radu Lacraru and Matthew Ho, whose MEng projects formed the engineering foundation that this project was built upon.

This work was supported by funding from Gardner’s UKRI fellowship ‘Verified Trustworthy Software Specification’ and grants from Meta and Amazon.

Bibliography

- [1] Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): *Deductive Software Verification – The KeY Book*, Lecture Notes in Computer Science, vol. 10001. Springer International Publishing, Cham (2016), ISBN 978-3-319-49811-9 978-3-319-49812-6, <https://doi.org/10.1007/978-3-319-49812-6>
- [2] Astrauskas, V., Bílý, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., Summers, A.J.: The prusti project: Formal verification for rust (invited). In: *NASA Formal Methods (14th International Symposium)*, pp. 88–108, Springer (2022), URL https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5
- [3] Ayoun, S., Denis, X., Maksimović, P., Gardner, P.: A hybrid approach to semi-automated rust verification. *Proc. ACM Program. Lang.* **9**(PLDI), 970–992 (2025), <https://doi.org/10.1145/3729289>, URL <https://doi.org/10.1145/3729289>
- [4] Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pp. 209–224, OSDI’08, USENIX Association, USA (Dec 2008)
- [5] Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, Lecture Notes in Computer Science, vol. 2988, pp. 168–176, Springer (2004), ISBN 3-540-21299-X
- [6] Eilers, M., Müller, P.: Nagini: a static verifier for python. In: *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I* 30, pp. 596–603, Springer (2018)
- [7] Ernst, G., Blau, J., Murray, T.: Deductive verification via the debug adapter protocol. *Electronic Proceedings in Theoretical Computer Science* **338**, 89–96 (Aug 2021), ISSN 2075-2180, <https://doi.org/10.4204/eptcs.338.11>, URL <http://dx.doi.org/10.4204/EPTCS.338.11>
- [8] Fragoso Santos, J., Maksimović, P., Ayoun, S.E., Gardner, P.: Gillian, part I: A multi-language platform for symbolic execution. In: *Programming Language Design and Implementation, PLDI (2020)*, ISBN 9781450376136, <https://doi.org/10.1145/3385412.3386014>, URL <https://doi.org/10.1145/3385412.3386014>
- [9] Hentschel, M., Bubel, R., Hähnle, R.: The symbolic execution debugger (sed): a platform for interactive symbolic execution, debugging, verification and more. *International Journal on Software Tools for Technology Transfer* (2019), ISSN 1433-2787, <https://doi.org/10.1007/s10009-018-0490-9>, URL <https://doi.org/10.1007/s10009-018-0490-9>
- [10] Holter, K., Hennoste, J.O., Saan, S., Lam, P., Vojdani, V.: Abstract debugging with gobpie. In: *Proceedings of the 2nd ACM International Workshop*

- on Future Debugging Techniques, p. 32–33, DEBT 2024, Association for Computing Machinery, New York, NY, USA (2024), ISBN 9798400711107, <https://doi.org/10.1145/3678720.3685320>, URL <https://doi.org/10.1145/3678720.3685320>
- [11] Jacobs, B.: `verifast-vscode` (2023), <https://github.com/verifast/verifast-vscode> [Accessed 2025/10/17]
 - [12] Karmios, N., Élie Ayoun, S., Gardner, P.: Gillian debugging: Swinging through the (compositional symbolic execution) trees, extended version (2026), URL <https://arxiv.org/abs/2602.07742>
 - [13] Karmios, N., Ayoun, S.E., Gardner, P.: [Artifact] Gillian Debugging: Swinging through the (Compositional Symbolic Execution) trees (Jan 2026), <https://doi.org/10.5281/zenodo.18311740>, URL <https://doi.org/10.5281/zenodo.18311740>
 - [14] Lööw, A., Nantes-Sobrinho, D., Ayoun, S.E., Cronjäger, C., Maksimović, P., Gardner, P.: Compositional Symbolic Execution for Correctness and Incorrectness Reasoning. In: Aldrich, J., Salvaneschi, G. (eds.) 38th European Conference on Object-Oriented Programming (ECOOP 2024), Leibniz International Proceedings in Informatics (LIPIcs), vol. 313, pp. 25:1–25:28, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2024), ISBN 978-3-95977-341-6, ISSN 1868-8969, <https://doi.org/10.4230/LIPIcs.ECOOP.2024.25>, URL <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.25>
 - [15] Lööw, A., Park, S.H., Nantes-Sobrinho, D., Ayoun, S.E., Sjöstedt, O., Gardner, P.: Compositional symbolic execution for the next 700 memory models. *Proc. ACM Program. Lang.* **9**(OOPSLA2) (Oct 2025), <https://doi.org/10.1145/3763151>, URL <https://doi.org/10.1145/3763151>
 - [16] Maksimović, P., Ayoun, S.E., Santos, J.F., Gardner, P.: Gillian, part ii: Real-world verification for javascript and c. In: *Computer Aided Verification (CAV)* (2021), ISBN 978-3-030-81687-2, https://doi.org/10.1007/978-3-030-81688-9_38, URL https://doi.org/10.1007/978-3-030-81688-9_38
 - [17] Meta: Infer Static Analyzer (2025), <https://fbinfer.com/> [Accessed 2025/10/17]
 - [18] Microsoft: Visual Studio Code (2024), <https://code.visualstudio.com> [Accessed 2025/10/17]
 - [19] Microsoft: Debug Adapter Protocol (2025), <https://microsoft.github.io/debug-adapter-protocol/> [Accessed 2025/10/17]
 - [20] Microsoft: Language Server Protocol (2025), <https://microsoft.github.io/language-server-protocol/> [Accessed 2025/10/17]
 - [21] Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) *Verification, Model Checking, and Abstract Interpretation*, vol. 9583, pp. 41–62, Springer Berlin Heidelberg, Berlin, Heidelberg (2016), ISBN 978-3-662-49121-8 978-3-662-49122-5, https://doi.org/10.1007/978-3-662-49122-5_2

- [22] O’Hearn, P.: Separation logic. *Commun. ACM* **62**(2), 86–95 (jan 2019), ISSN 0001-0782, <https://doi.org/10.1145/3211968>, URL <https://doi.org/10.1145/3211968>
- [23] Panić, S.: Collections-C (2025), <https://github.com/srdja/Collections-C> [Accessed 2025/10/17]
- [24] Raad, A., Berdine, J., Dang, H.H., Dreyer, D., O’Hearn, P., Villard, J.: Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In: Lahiri, S.K., Wang, C. (eds.) *Computer Aided Verification*, pp. 225–252, Springer International Publishing, Cham (2020), ISBN 978-3-030-53291-8, https://doi.org/10.1007/978-3-030-53291-8_14
- [25] The Infer Team @ Meta: Infer Pulse (2025), <https://fbinfer.com/docs/checker-pulse> [Accessed 2025/10/17]
- [26] Wolf, F.A., Arqunt, L., Clochard, M., Oortwijn, W., Pereira, J.C., Müller, P.: Gobra: Modular specification and verification of go programs (extended version) (2021)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





RECHECK: Automated Contextual Improvement Verifier for Functional Calculi across User-Defined Operational Semantics

Makoto Hamana¹ and Kento Emoto¹

Department of Computer Science and Networks, Kyushu Institute of Technology
`{hamana,emoto}@csn.kyutech.ac.jp`

Abstract. Contextual improvement is a directed form of contextual equivalence that ensures a reduction in the number of evaluation steps. We present RECHECK, an automated tool that verifies contextual improvement for foundational calculi (such as extensions of the λ -calculus) and higher-order functional programs across operational semantics defined by Felleisen’s evaluation contexts. RECHECK allows users to define syntax, evaluation, and refinement rules as rewriting systems called Term Evaluation and Refinement Systems (TERS). We implement the theory of TERS and the method for deriving contextual improvement proposed by Muroya and Hamana in the tool RECHECK. We demonstrate RECHECK’s applicability by proving contextual improvement across diverse calculi and programs.

1 Introduction

Contextual equivalence [23] is a fundamental concept in programming language theory, capturing the idea that two program fragments are indistinguishable in all contexts under a given operational semantics. Contextual improvement [29] provides a directed alternative, ensuring a reduction in evaluation steps. This notion is crucial for program optimization, where transformations must be both semantically valid and efficiency-preserving. Despite its significance, proving contextual improvement remains a complex and manual process, relying on pen-and-paper proofs that are tedious and error-prone. Recently, we proposed a theory to address this problem based on a novel rewriting theory of *term evaluation and refinement systems* (TERS), which enables systematic reasoning about evaluation and refinement rules [25]. Within the theory of TERS, we have given sufficient conditions to lead contextual improvement. It involves checking various conditions. The most complex part is *critical pair analysis*, which requires enumerating and checking conflicts between rewrite rules, originally developed in rewriting theory for establishing the confluence property [19, 1]. Although this checking is decidable, it is tedious and error-prone when done manually. The complexity increases significantly as the number of rules grows. In this paper, we introduce a new tool RECHECK, a Haskell-based verification tool that automates the contextual improvement verification.

Organization. The rest of the paper is structured as follows. Section 2 provides an overview of our approach, illustrating how contextual improvement can be formulated and verified using rewriting techniques, and how the tool **ReCHECK** automates it. Section 3 describes the design and implementation of the **ReCHECK** tool. Section 4 presents experimental results on foundational calculi and functional-program benchmarks. Section 5 compares **ReCHECK** with existing verification tools and discusses its limitations and future work.

2 Proving Contextual Improvements in the TERS framework: Rewriting Theory and **ReCHECK**

We explain the theory [25] and how the tool **ReCHECK** automates that sample refinement rules are contextual improvements.

2.1 Rewriting theory

We consider the call-by-value λ -calculus extended with the Maybe monad [36, 22] as used in Haskell. The Maybe monad handles computations that may fail by explicitly representing values as **Just** a (success) or **Nothing** (failure). Using the bind ($\gg=$), operations chain safely, skipping on **Nothing**. It provides pure, side-effect-free failure handling with explicit failure representation. A monad **Maybe** is a structure with two operations $\text{return} : a \rightarrow \text{Maybe}(a)$, $\gg= : \text{Maybe}(a) \rightarrow (a \rightarrow \text{Maybe}(b)) \rightarrow \text{Maybe}(b)$ satisfying the three monad laws. We define a typed λ -calculus extended with the Maybe monad by the *evaluation rules* $\mathcal{E}$

$$\begin{aligned}
 \lambda(x.M[x])@V &\Rightarrow M[V] & (\beta) \\
 \text{Nothing} \gg= x.F[x] &\Rightarrow \text{Nothing} & (\text{bi1}) \\
 \text{Just}(M) \gg= x.F[x] &\Rightarrow F[M] & (\text{bi2}) \\
 \text{return}(M) &\Rightarrow \text{Just}(M) & (\text{ret})
 \end{aligned}$$

where we use the symbols λ for λ -abstraction and $@$ for applications. To clarify the distinction of free and bound variables, we write the free variables in capitals, and maintain bound variables as small letters. We use the square brackets $M[x]$ to denote an application of a term to a free variable M . The notation $M[V]$ is ordinarily written as $M[x := V]$, meaning that the variable x appearing in M is replaced with V . In fact, this is the notation of second-order algebraic theory [6, 10, 13]). The (β) is the call-by-value β -reduction rule and the rest are the ordinary definition of the Maybe monad. To make the rewrite system an operational semantics, we define Felleisen-style evaluation contexts E [3]:

$$E ::= \square \mid \text{return}(E) \mid \text{Just}(E) \mid E@t \mid V@E \mid E \gg= t$$

with arbitrary terms t and values $V ::= \lambda(x.t) \mid \text{Just}(V) \mid \text{Nothing}$. The evaluation relation $\rightarrow_{\mathcal{E}}$ is generated by the set $\mathcal{E}$ of evaluation rules closed under

the evaluation contexts. We consider the monad laws as rewrite rules called **refinement rules** $\mathcal{R}$.

$$\text{return}(N) \gg= y. M[y] \Rightarrow M[N] \quad (\text{idL})$$

$$M \gg= y. \text{return}(y) \Rightarrow M \quad (\text{idR})$$

$$(M \gg= x. K[x]) \gg= y. L[y] \Rightarrow M \gg= x. (K[x] \gg= y. L[y]) \quad (\text{assoc})$$

A scenario in which this could be used as a refinement would be as follows. If a term in a program matches with the left-hand side of one of the above rules, then the compiler rewrites it to the corresponding right-hand side at compile time. This should be theoretically allowed by the fact that the Maybe monad satisfies the monad laws.

Rewriting theory for deriving contextual improvement. We show that the monad laws can be safely used for optimizing a program with the Maybe monad following the rewriting theoretic method developed in [25]. The pair $(\mathcal{E}, \mathcal{R})$ constitute a **term evaluation and refinement system (TERS)**. From it, we obtain evaluation $\rightarrow_{\mathcal{E}}$ and refinement $\Rightarrow_{\mathcal{R}}$ on well-typed meta-terms defined by

$$\frac{(l \rightarrow r) \in \mathcal{E} \quad E \in Ectx}{E[l\theta] \rightarrow_{\mathcal{E}} E[r\theta]} \quad \frac{(l \Rightarrow r) \in \mathcal{R} \quad C \in Ctx}{C[l\theta] \Rightarrow_{\mathcal{R}} C[r\theta]}$$

where θ is a substitution, $Ectx$ is the set of all evaluation contexts and Ctx is the set of all ordinary contexts. This means that evaluation is only closed under evaluation contexts to make it operational semantics, whereas the refinement is closed under arbitrary context to optimize arbitrary program fragments.

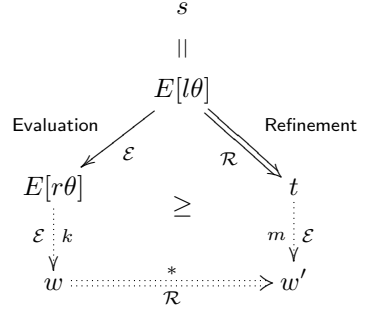
The central problem in this paper is contextual improvement. A set $\mathcal{R}$ of refinement rules is a **contextual improvement** w.r.t. a set $\mathcal{E}$ of evaluation rules if the following condition is satisfied. Suppose a refinement rewrite step $s \Rightarrow_{\mathcal{R}} t$. Then for any context C such that $C[s]$ and $C[t]$ are closed terms, $C[s] \xrightarrow{k}_{\mathcal{E}} v$ implies $C[t] \xrightarrow{m}_{\mathcal{E}} v'$ such that $v =_{Val} v'$, where v is a value and k and m are the number of evaluation steps such that $k \geq m$ and $=_{Val}$ is an equality on values. This means that the refinement preserves the value v and actually improves the efficiency by reducing the number of evaluation steps. Since refinement is closed under contexts, it is depicted as the following diagram. In general, it is difficult to prove this, because there is universal quantification over all contexts, and it would also be difficult to deal with the evaluation steps and guarantee its decreasing. We have proved the following key theorem.

$$\begin{array}{ccc} C[s] & \xRightarrow{\mathcal{R}} & C[t] \\ \swarrow \varepsilon_k & & \searrow \varepsilon_m \\ v & \xRightarrow{=_{Val}} & v' \end{array}$$

Theorem 21 ([25, Thm. 25]) *Let $(\mathcal{E}, \mathcal{R})$ be a TERS. If (1) $\mathcal{E}$ is deterministic, (2) $\mathcal{R}$ is value-invariant and (3) $(\mathcal{E}, \mathcal{R})$ is locally coherent, then refinement $\mathcal{R}$ is a contextual improvement w.r.t. evaluation $\mathcal{E}$.*

This requires to check conditions (1)-(3). We check these for the Maybe monad. The condition (1) of determinism means that the evaluation relation $\rightarrow_{\mathcal{E}}$ is deterministic (i.e. if $t_1 \leftarrow_{\mathcal{E}} s \rightarrow_{\mathcal{E}} t_2$ then $t_1 = t_2$), which is ensured by the evaluation contexts. The condition (2) of value-invariant means that a refinement of a value is again a value. This is always satisfied in ordinary functional programming languages because values are usually not rewritten by the refinement rules. The condition (3) of **local coherence** [25] is the most important and involved to check. It is depicted as the situation (♦). It means the following: suppose a given term s is rewritten in two ways — by evaluating to $E[r\theta]$ (using a rule $l \rightarrow r \in \mathcal{E}$ with substitution θ , within an evaluation context E) and by using refinement $\mathcal{R}$ to t , respectively. Then, through further evaluation, they reach w and w' (which are not necessary values) in k - and m -steps, respectively. Moreover, the term w must be refined to w' by the many-step refinement rewriting $\Rightarrow_{\mathcal{R}}^*$ (including zero steps). Local coherence is a condition that for any term s , this situation is satisfied and the evaluation steps are decreasing as (the original steps =) $1 + k \geq m$ (= the evaluation steps after refinement).

(♦) **Local coherence**



Checking all these conditions for all terms in every situation appears to be difficult. Essentially, this represents a conflict between evaluation and refinement. Fortunately, a method of analysis of similar conflicts can be found in rewriting theory. It is called *critical pair analysis* used in the proof method of deriving confluence of rewrite systems [1]. Critical pairs have been used for a different purpose — namely, to show *local confluence*, which is a restricted version of the confluence property. It is a method to enumerate all the conflict rewrites between two overlapped rules and checking their joinability. The “critical pair lemma” concludes that it derives local confluence.

Local coherence can be deduced by calculating *critical pairs*—regarding the two rules as a conflict between evaluation and refinement—and systematically verifying all such cases. This is suitable for automation. This relies on the “critical pair lemma”.

Lemma 22 ([25, Thm. 13]) *A well-behaved TERS $(\mathcal{E}, \mathcal{R})$ is locally coherent if and only if all its critical pairs are joinable.*

Definition 23 [25, Def. 12] A TERS $(\mathcal{E}, \mathcal{R})$ is called *well-behaved* if it satisfies:

- (i) For any $E \in Ectx$ and $C \in Ctx$, if $E \Rightarrow_{\mathcal{R}} C$ then $C \in Ectx$.
- (ii) Every $(l \rightarrow r) \in \mathcal{R}$ satisfies: both l and r are linear w.r.t. non-value metavariables. For every non-value metavariable M , if $l[M \mapsto \bar{x}.\square] \in Ectx$ then $r[M \mapsto \bar{x}.\square] \in Ectx$.
- (iii) For any $(l \rightarrow r) \in \mathcal{E}$, l is linear w.r.t. non-value metavariables.

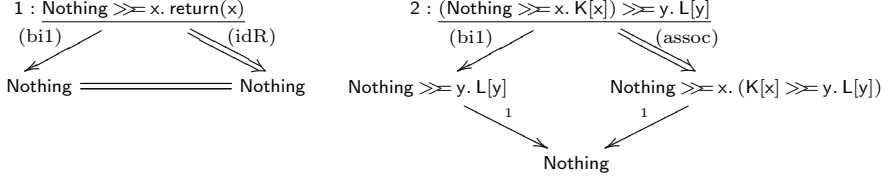


Fig. 1. Critical pairs of the Maybe monad (excerpt)

Checking critical pairs for local coherence. Let us now return to the analysis of the Maybe monad, and see how critical pairs are useful by using Lemma 22. A critical pair is a situation that admits two ways of rewriting by evaluation and refinement, respectively. Careful inspection of $\mathcal{E}$ and $\mathcal{R}$ reveals that it has, in all, 5 patterns of critical pairs. In Fig. 1, we excerpt two patterns, each of which admits the evaluation $\rightarrow$ and the refinement $\Rightarrow$.

The pair of divergent terms is called a *critical pair*. The above diagrams show that all the critical pairs are joinable by further evaluation. For example, the critical pair 2: shows an overlap between the rules (bi1) and (assoc). Both rules' left-hand sides become the same top term ($\text{Nothing} \gg= x.K(x)$) $\gg= y.L[y]$, which is obtained by applying the unifier $M \mapsto \text{Nothing}$, $F \mapsto x.K(x)$ to the left-hand sides. The top term is evaluated (by $\rightarrow$) to $\text{Nothing} \gg= y.L[y]$ and refined (by $\Rightarrow$) to $\text{Nothing} \gg= x.(K[x] \gg= y.L[y])$. Both terms are evaluated to Nothing by 1 (left term) and 1 (right term) further evaluation steps, respectively. As the conditions of local coherence, the condition (the original evaluated steps =) $1 + k \geq m$ (= the evaluation steps after refinement) must be satisfied. In this case, it holds as $1 + 1 \geq 1$, hence the critical pair is joinable. Other cases are similar. As this example shows, critical pairs are calculated by computing the most general unifiers between the left-hand sides of rules. To apply Lemma 22, we need also to check the condition of “well-behavedness” consisting of various syntactic conditions of rule sets. The Maybe monad could be shown to be well-behaved. We conclude that every rule in $\mathcal{R}$ is a contextual improvement.

2.2 Verifying contextual improvement by RECHECK

As the Maybe monad example shows, the most complex part is to exhaustively enumerate all critical pairs and check their joinability. One must try to do the following:

- (i) enumerate all the overlaps of the rules without oversight
- (ii) rewrite terms to check joinability with counting the number of evaluation steps
- (iii) checking various conditions of well-behavedness of rules

These are often quite confusing and prone to error because of similar rules, terms, and variable names. Moreover, one often needs trial and error to develop better refinement rules. It often happens that the rule given is not actually an improvement: possible reasons for not joining are

- (i) the rule is wrong (e.g. not preserving values),
- (ii) more refinement rules are required to make them contextual improvement, or
- (iii) the critical pairs are joinable but the number of evaluation steps is not decreasing (i.e., it is not an optimisation).

In order to devise better refinement rules in such cases, a tool is needed to develop and test new rules quickly. Therefore, we develop a tool **RECHECK** that automatically checks all the conditions to derive local coherence.

A usage scenario for RECHECK. We show how proofs can be completed by the tool **RECHECK**. Our system **RECHECK** is implemented as an embedded domain specific language (DSL) using Template Haskell. The user creates a Haskell script for a given problem by specifying a signature and rules. Then in the Glasgow Haskell interpreter (GHCi), the user attempts several commands to analyse local coherence. The command is realised as a Haskell function. The GHCi interpreter provides an interactive user interface for **RECHECK**.

Specifying Maybe monad in RECHECK. First, we define the type signature for the Maybe monad:

```
maybeSig = [signature|
  return : a -> Maybe(a); bind : Maybe(a), (a -> Maybe(b)) -> Maybe(b)
  nothing : Maybe(a);      just : a -> Maybe(a)  |]
```

Then we define the evaluation rules corresponding to $\mathcal{E}$ by

```
maybeP = [rules|
  (beta) lam(y.M[y]) @ V    => M[V]; (bi1) nothing >>= x.F[x] => nothing
  (bi2) just(M) >>= x.F[x] => F[M]; (ret) return(M)           => just(M)
  |]
```

and the refinement rules corresponding to $\mathcal{R}$.

```
maybeR = [rules|
  (idL)  return(N) >>= y.M[y] => M[N]; (idR) M >>= y.return(y) => M
  (assoc) (M >>= x.K[x]) >>= y.L[y] => M >>= x.(K[x] >>= y.L[y]) |]
```

We use these as schemas of computation. Therefore, free variables are important as the targets of instantiation by pattern matching. The keyword-headed bracket `[signature|...]` or `[rules|...]` indicates the beginning and end of **RECHECK**'s DSL using a feature of Template Haskell.

Values and evaluation contexts by order sorts. We need to define values and evaluation contexts for formulation of TERS. The evaluation rule is not a simple rewrite rule; rather, it is applied within an evaluation context. Moreover, when formalizing values and expressing call-by-value semantics, it becomes necessary to distinguish between “value variables” (which can only be replaced with values) and general variables (which are replaced with arbitrary terms).

This distinction defines *syntax classes* distinct from types. In [25], we formalized TERS as a formal system whose components include such syntactic classes.

In this paper, to implement RECHECK, we refine the original formalization of TERS. Specifically, we employ the notion of *order-sorts* for syntactic classes. The concept of order-sorts was originally introduced in the specification language OBJ [9] and later used to express inheritance relations in object-oriented systems. Syntactic classes such as values must be represented as sorts distinct from types. To this end, we introduce a sort v to represent values, corresponding to the syntactic class “values.” There is also a sort t for all terms, and since values form a subset of terms, we assume the subsort relation $v \sqsubseteq t$. This is precisely why we employ order-sorts.

Rewriting in TERS is formalized so that substitution is applied in a manner that respects this order-sort structure. Furthermore, order-sorts are useful in formalizing evaluation contexts. An evaluation context is typically defined in BNF as a subclass of all contexts. By defining it as a term with the sort e (standing for “evaluation context”), we can naturally formulate the terms of evaluation uniformly. Moreover, the check required during rewriting – namely, whether rewriting occurs within an evaluation context – can be implemented as a sort check. Practically, this sort checking can reuse the existing type checker, thereby contributing to an efficient implementation.

Following this methodology, we illustrate the case of the Maybe monad. We have defined values as $V ::= \lambda(x.t) \mid \text{Just}(V) \mid \text{Nothing}$. In RECHECK, this is declared by specifying function symbols having the target sort v .

```
maybeValues = [signature| abs : t -> v; nothing : v; just : v -> v ]
```

Likewise, according to the BNF definition of E , we declare function symbols for evaluation contexts,

```
maybeCtx = [signature| xhole : e; return : e -> e; just : e -> e ...]
```

thereby giving an appropriate definition of terms that constitute evaluation contexts.

Checking local coherence. The tool RECHECK provides automatic checking of local coherence. Invoking the command `improve`, RECHECK first checks the necessary conditions of *well-behavedness* on the rules. Then, RECHECK enumerates all critical pairs as shown in Fig.1, and checks its joinability while rigorously counting the number of evaluations from both sides of each critical pair. It identifies overlaps by applying *higher-order unification* [17, 21], which allows subterms in the left-hand side of one rule to be matched with the root of another. Each critical pair thus obtained corresponds to a divergent peak in the diagrams in Fig.1, where the source term is unified and instantiated accordingly. Using this information, RECHECK then performs a joinability test by exhaustively rewriting both branches and comparing the results. The number of steps taken along each branch is explicitly tracked, and the outcome is either syntactic equality or equivalence modulo refinement steps. This mechanized check ensures that

all critical pairs are locally joinable in the precise sense required by contextual improvement. Therefore, by Theorem 21, we conclude that the rule set $\mathcal{R}$ constitutes a contextual improvement over $\mathcal{E}$.

3 Tool RECHECK

Our tool RECHECK is written in Haskell consisting of about 15000 lines of code. It consists of various theory-based components. We discuss each component in turn.

Second-order abstract syntax and DSL RECHECK's specification language based on second-order abstract syntax [7] with metavariable [10, 4] and polymorphism [11, 5]. Second-order abstract syntax cleanly provides the general syntax to describe operational semantics of higher-order languages. It is provided as a domain specific language (DSL), implemented in Template Haskell [31], for writing and modifying TERS specifications interactively. When an automatically checked refinement fails, the user can refine or extend the specification – for example, by adding auxiliary refinement rules analogous to completion steps in rewriting – and immediately re-run verification. This design makes RECHECK not only a verifier but also an exploratory environment for semantic reasoning.

Second-order rewriting engine for evaluation and refinement

Evaluation and refinement of TERS are based on polymorphic second-order rewriting [13, 14]. Therefore, we provide second-order rewriting engine by extending the rewriting engine of the previous tool of second-order laboratory SOL [13, 14]. The necessary extension is to respect evaluation contexts. This is implemented by introducing order sorts described in §2.2.

Higher-order pattern unification and matching Second-order rewriting engine requires higher-order pattern matching, which is decidable by imposing Miller's higher-order patterns for the left-hand sides [21]. Moreover, to compute critical pairs, higher-order pattern unification is necessary. This is implemented by following the functional implementation of higher-order pattern unification [12] in Haskell. Pattern matching is then obtained as a special case of unification.

Critical pair enumeration and joinability tests There are two points to note that differ from the usual critical pair analysis in rewriting theory: first, critical pairs are not always possible to generate from overlaps. This requires checking an evaluation context by sort checking. Another point is the joinability test. The number of evaluations is counted and it is checked that it decreases. Importantly, we do not assume termination of evaluation, so the idea of implementing the joinability test as a comparison of normal forms cannot be used. In the implementation of RECHECK, we limit the joinability test to a certain number of evaluations and terminate it after that. This is reasonable because it is expected that the result (or a term

appears on the way to the result) of the refinement will be the same as the original, and it is unlikely that the number of evaluations will be extremely long if the results are the same. This does not affect the soundness. Since the evaluation is deterministic, we only need to look at one reduction path for each evaluation going down for joinability.

Well-behavedness and value-invariance The conditions of well-behavedness of TERS (Def. 23) are checked by directly checking syntactic conditions, such as linearity of variables. Importantly, sort checking is also effectively used to implement the conditions of preservation of evaluation contexts in refinement rules.

Determinism There are two sources of non-deterministic rewriting. (1) when there is more than one rule to match, and (2) the choice of multiple redexes in a term. For (1), we check a condition — known as *orthogonality* [32, Sec.4] in rewriting theory — that there is no overlap between the evaluation rules and that they are left linear. For (2), it is sufficient to check that the evaluation contexts are appropriately defined.

4 Case Studies and Benchmarks

To evaluate the effectiveness of RECHECK, we conducted an extensive benchmark study. Firstly, we consider three examples of foundational calculi for functional programming languages, and demonstrate two aspects of RECHECK: (1) the flexibility of the formalism that faithfully represents different calculi involving own syntax and data: values, monadic computation, let-binding, and λ -bindings, and (2) the power of automation. Secondary, we consider functional program examples and try to compare existing functional program verifiers.

4.1 Simplified computational λ -calculus λ_{ml*}

We consider Sabry and Wadler’s computational λ -calculus λ_{ml*} [28, 2], an extension of the simply-typed λ -calculus with **let** and **return** for monadic computation. It forms a TERS [25, Ex.18]. We formalise it in RECHECK and show automatic verification. It has the values, computations and evaluation contexts:

Values $V, V' ::= x \mid \lambda(x.P)$ Evaluation contexts $E ::= \square \mid \text{let}(E, x.P)$
 Computations $P, P' ::= \text{return}(V) \mid \text{let}(P, x.P') \mid V@V'$

We use the three sorts: values **v**, computations **p**, and terms **t**. We declare the sort declaration **clamSort** for the syntax reflecting the above BNF:

```
clamSort = [signature| lam : t -> v;   xhole : e; let : e, p -> e
  return : v -> p; let : p, p -> p; app : v, v -> p ..]
```

Note that the original let-syntax **let** $x = s$ **in** t is represented as a meta-term **let** $(s, x.t)$, where $x.-$ is a variable binding, which is the notation of the second-order abstract syntax [7, 10, 4, 11]. We also declare variables used in the rules to

have appropriate sorts, i.e., V for values of sort v and P for computations of sort p .

```
clamVars = [variables| V : v; P : p |]
```

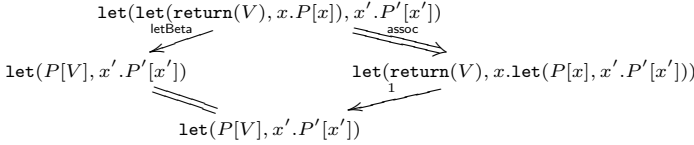
The λ_{ml*} -calculus consists of 5 rules. We divide it into the evaluation rules **clamP** and the refinement rules **clamR**

```
clamP = [rules|
  (beta)  lam(x.P[x])@V => P[V]; (letBeta) let(return(V),x.P[x]) => P[V]
  |]
clamR = [rules|
  (eta)   lam(x.V@x) => V;          (letEta) let(P, x.return(x)) => P
  (assoc) let(let(P1,x.P2[x]),y.P3[y]) => let(P1,x.let(P2[x],y.P3[y])) |]
```

Now, we consider whether **clamR** is a contextual improvement. From their forms, these rules can be regarded not as computations but rather as optimizations or reorganizations. We invoke the command

```
ReCheck> improve clamSort clamVars clamP clamR
```

in GHCi, which means to check contextual improvement of **clamR** w.r.t. **clamP** under the sort signature **clamSort** with sorted variables **clamVars**. **RECHECK** firstly checks the necessary syntactic conditions on TERS (determinism, value-invariance, and well-behavedness). Then it automatically enumerates 3 critical pairs, such as the critical pair caused by the overlap between **assoc** and **letBeta**:



RECHECK confirms that all the critical pair are joinable with counting the numbers of evaluation steps and checking their decreasing. We conclude that this TERS a contextual improvement. This fact has not been known in the literature [28, 2].

4.2 Effect handlers

We consider a larger example: a calculus for effect handlers [26] as a TERS [25, Ex. 19]. We consider two operations, op_1 and op_0 , and handlers: **handler₁**, which catches the first operation op_1 , and **handler₀**, which catches no operation. This TERS has more syntactic classes than the λ_{ml*} :

Functions $F ::= x \mid \text{fun}(x.P)$ Values $V ::= \text{true} \mid \text{false} \mid F \mid H$

Eval. contexts $E ::= \square \mid \text{do}(E, x.P) \mid \text{with}(H, E) \mid \text{if}(E, t, t) \mid E @ M \mid V @ E$

Handlers $H ::= \text{handler}_1(x.P, x.k.P_1) \mid \text{handler}$

Computations $P ::= \text{return}(V) \mid \text{op}_1(V, y.P) \mid \text{op}_0(V, y.P) \mid \text{do}(P_1, x.P_2)$

We use the sort **f** for functions, sort **v** for values, and sort **e** for evaluation contexts and declare the sort declaration **clamSort** as

```
handlerSort = [signature|
  fun : p -> f; fun : p -> v; handler1 : p,p -> h; handler0 : p -> h
  op1 : v,p -> p ; op0 : v,p -> p; with : h,p -> p; ... |]
```

We divide the original rules [26] into **handlerP** and **handlerR** (where $i = 0, 1$):

```
handlerP = [rules|
  (1) do(return(V), x.P[x]) => P[V]; (5) fun(x.P[x])@V => P[V]
  (2i) do(opi(V, y.P1[y]), x.P2[x]) => opi(V, y.do(P1[y], x.P2[x]))
  (3) if(true, P1, P2) => P1; (4) if(false, P1, P2) => P2
  (6) with(handler1(x.P[x], x.k.P1[x,k]), return(V)) => P[V]
  (7) with(handler1(x.P[x], x.k.P1[x,k]), op1(V, y.P'[y])) => P1[V, fun(y.P'[y])]
  (8) with(handler1(x.P[x], x.k.P1[x,k]), op0(V, y.P'[y]))
      => op0(V, y.with(handler1(x.P[x], x.k.P1[x,k]), P'[y]))
  (9) with(handler0(x.P[x]), return(V)) => P[V]
  (10i) with(handler0(x.P[x]), opi(V, y.P'[y])) => opi(V, y.with(handler0(x.P[x]), P'[y])) |]

handlerR = [rules|
  (r3) do(P, x.return(x)) => P; (r9) fun(x.F@x) => F
  (r4) do(do(P1, x.P2[x]), x.P3[x]) => do(P1, x1.do(P2[x1], x2.P3[x2]))
  (r13) with(handler0(x.P[x]), P') => do(P', x.P[x]) |]
```

The evaluation rules **handlerP** define the operational semantics of effect handlers. In contrast, the rules **handlerR** are refinement rules: they express program transformations that simplify or reorganize handler expressions while preserving their meaning. For example, rule (r3) eliminates a redundant **do**-expression, and (r13) replaces a handler call by an equivalent sequencing form.

We verify that **handlerR** is a contextual improvement. As the number of rules suggests, critical pair analysis in this case requires many overlap checking. For each rule in **handlerR**, overlaps with the ten **handlerP** rules are computed by higher-order pattern unification with *sort checking* because when overlaps occur, the critical pairs and their joinability *with evaluation-step counting* must be verified. Invoking the **improve** command, RECHECK automatically enumerates *10 critical pairs* and confirms their joinability, such as

$$\begin{array}{ccc}
 & \text{do}(\text{do}(\text{op}_i(V, x.P[x]), y.P_2[y]), z.P_3[z]) & \\
 \swarrow 2_i & & \searrow r_4 \\
 \text{do}(\text{op}_i(V, x.\text{do}(P[x], y.P_2[y])), z.P_3[z]) & & \text{do}(\text{op}_i(V, x.P[x]), y.\text{do}(P_2[y], z.P_3[z])) \\
 \downarrow & & \downarrow \\
 \text{op}_i(V, x.\text{do}(\text{do}(P[x], y.P_2[y]), z.P_3[z])) & \xRightarrow{r_4} & \text{op}_i(V, x.\text{do}(P[x], y.\text{do}(P_2[y], z.P_3[z])))
 \end{array}$$

It is a contextual improvement. This fact has not been shown in the literature [26]. Moreover, this also underscores the necessity of automated verification by RECHECK. Given the large number of critical pairs that might otherwise be overlooked in manual analysis.

4.3 Call-by-need λ -calculus

We consider a call-by-need λ -calculus by Maraist, Odersky, Turner, and Wadler [20], reformulated as a TERS [24]. It has 4 syntactic classes assigning sorts **a**, **v**, **r**, **e**:

Answers $A ::= x \mid \lambda(x.M)$ Values $V ::= \lambda(x.M) \mid \text{let}(M, x.V)$
 Residuals $R ::= R@t \mid \text{let}(t, r.R)$
 Evaluation context $E ::= \square \mid E@M \mid \text{let}(M, x.E) \mid \text{let}(E, x.R)$

The call-by-need λ -calculus consists of four evaluation rules:

```

needP = [rules|
  (I)  $\text{lam}(x.M[x])@N \Rightarrow M[N]$ ; (C)  $\text{let}(A, x.R[x]) \Rightarrow R[A]$ 
  (V)  $\text{let}(M, x.V[x])@N \Rightarrow \text{let}(M, x.V[x]@N)$ 
  (A)  $\text{let}(\text{let}(M, x.V[x]), y.R[y]) \Rightarrow \text{let}(M, x.\text{let}(V[x], y.R[y]))$   |]

```

This example demonstrates the evaluation generality of RECHECK. It can handle a computation system based on call-by-need, which is neither call-by-value nor call-by-name. The formalization of call-by-need was clarified in works such as [20], where the evaluation is properly controlled by means of let-expressions. RECHECK is capable of formalizing and verifying such a complex evaluation strategy. We take the refinement rules **needR** to be the same **needP** extended with (**eta**) rule

```

needR = needP ++ [rules| (eta)  $\text{let}(M, x.N) \Rightarrow N$  |]

```

We verify that **needR** is a contextual improvement. Invoking RECHECK, it checks the necessary conditions of TERS and then enumerates two critical pairs and confirms their joinability. Therefore, **needR** is a contextual improvement.

4.4 Lazy program involving infinite lists

We consider a lazy program

```

takeP = [rules|
  (rep1)  $\text{replicate}(z) \Rightarrow []$  ; (take1)  $\text{take}(z, XS) \Rightarrow []$ 
  (rep2)  $\text{replicate}(s(N)) \Rightarrow s(z):\text{replicate}(N)$ ; (take2)  $\text{take}(s(N), []) \Rightarrow []$ 
  (ones)  $\text{ones} \Rightarrow s(z) : \text{ones}$  ; (take3)  $\text{take}(s(N), X:XS) \Rightarrow X : \text{take}(N, XS)$ 

```

This is considered as a Haskell program involving infinite lists. It defines basic list operations using Peano natural numbers. **replicate** generates a list of repeated **s(z)**'s, **take** extracts the first N elements from a list, and **ones** denotes an infinite list of ones. We consider a TERS having **takeP** as evaluation rules and the following refinement rule (**conj**)

```

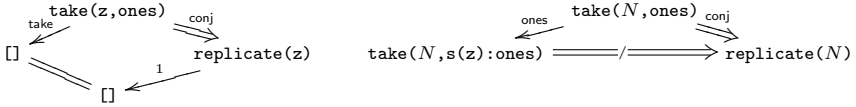
(conj)  $\text{take}(N, \text{ones}) \Rightarrow \text{replicate}(N)$ 

```

This means: since the left-hand side of (**conj**) takes N elements from **ones**, it should be valid to rewrite it into N copies of **s(z)**.

This problem was discussed in [18] as a problem of an *inductive theorem* of term rewriting [16, 27]. That is, the question is whether (**conj**) (considered as an equation) can be derived as a theorem if the program is regarded as a set of equational axioms. The problem is called an inductive theorem in the sense that not only equational logic but also induction can be used here. However, all

inductive theorem proving methods known so far have required that a given program is terminating or converges to values. Therefore, problems such as (`conj`) could not be proved by existing methods for rewriting systems. [18] proposed an innovative method for this, which does not assume termination. The problem, however, is that it assumes the condition called local sufficient completeness, which is often involved to derive [18]. In this context, it is interesting to consider whether our method can show this as improvement. RECHECK reports two critical pairs and one non-joinable.



This non-joinable pair is a semantically valid rewrite. Therefore, we regard it as a missing rule and incorporate it as an additional refinement rule:

(ext) `take(N,s(z):ones) => replicate(N)`

Checking again `takeR++ext`, RECHECK reports 4 critical pairs and all are joinable. In summary, RECHECK can also verify such lazy problems (with a little effort). This also illustrates another important use of RECHECK. It contributes to “better refinement rule design” by looking at the output even when it fails, rather than just as a verifier. Such “Knuth-Bendix completion [19]”-like method is effective in many cases.

4.5 Functional program benchmarks

As the previous example illustrates, RECHECK can also verify optimization rules for functional programs. This is possible because higher-order functional programs can be represented as TERSs using second-order abstract syntax, which is a simpler structure than fundamental calculi such as the call-by-need λ -calculus. Moreover, any evaluation strategy – including call-by-value, call-by-name, and their combinations – can be specified through evaluation contexts. This allows RECHECK to handle a wider range of programs than ordinary functional programming verifiers. Here we consider the following 9 functional program examples. These benchmarks were chosen to cover a variety of program structures and semantic phenomena – from first-order recursive functions on algebraic datatypes (`append` (`++`), `reverse`, `take`, `length`) to higher-order (`map`), lazy (`ones`) and non-terminating (`loop`) programs.

1. (`map/map`) `map(x.F[x],map(y.G[y],XS)) => map(a.F[G[a]], XS)`
2. (`map/append`) `map(x.F[x],XS) ++ map(x.F[x],YS) => map(x.F[x], XS ++ YS)`
3. (`assoc`) `(XS ++ YS) ++ ZS => XS ++ (YS ++ ZS)`
4. (`revrev`) `reverse (reverse XS) => XS`
5. (`lenPlus`) `length XS + length YS => length (XS++YS)`
6. (`ones`) `ns(1) => ones`

7. (refine) `loop => 0`
8. (copy) `copy (copy N) => N`
9. (make-list) `length (make-list M) => M`

Each rule listed above represents a candidate refinement between two higher-order functional program fragments. Whether these rules are indeed contextual improvements is not evident a priori, for instance, the fusion rule (`map/map`) or the associativity rule (`assoc`) could, in principle, alter program behavior depending on the evaluation strategy. The key point is that RECHECK can automatically determine their contextual improvements through critical pair analysis. Several refinement rules require additional explanation.

- (`revrev`) expresses the involutive property of list reversal. This is a typical exercise of equation proved by induction requiring lemmas.
- (`ones`) states a refinement of two infinite lists of 1s, from concrete (`ones`) to abstract (`ns(N)`), where `ones => 1:ones`, `ns(N) => N:ns(N)`.
- (refine) where `loop` is defined by `loop => loop`. (refine) means that non-terminating `loop` is refined to the terminating constant.
- (`copy`) shows that the recursively defined identity function on naturals defined by `copy z = z`; `copy (s(X)) = s (copy X)` satisfies the idempotency.
- (`make-list`) `make-list` generates a list of a given length M .

Result. RECHECK successfully verifies that all these refinements as contextual improvements except for (`revrev`). As in §4.4, once the verification fails, but adding missing rules manually to the set of refinement rules, RECHECK could verify (`revrev`). This is also reflected in Table 1, which shows the result of trying the same problems (`revrev`) and (`conj`) twice.

4.6 Comparison with other tools for program verification

We are not aware of any other tool that automatically verifies contextual improvement for functional programs and calculi with user-defined operational semantics. As such, RECHECK is unique, making rigorous comparisons with other tools difficult. However, by restricting the scope of problems, ReCheck may be applicable to problems handled by other verification tools, making some comparisons meaningful. We consider three tools. Mochi [30, 34] is a verification tool for OCaml using constrained Horn clauses and automatically checks safety properties using SMT solving. RCaml [33, 35] is a verification tool for OCaml using refinement types that performs SMT-based verification of program invariants. Timbuk [15] is a verification tool based on tree-automata completion. All of them target call-by-value higher-order functional programs.

Setup. For comparison, we applied the examples used in this paper to the three tools MoCHi, RCaml, and Timbuk, when possible, recording whether each tool could verify the property and the verification time. Verification times were measured on an Intel Core i5 13th Gen. CPU with 16GB RAM. All results

were obtained using fully automatically without manual proof assistance. For RCaml, two different back-end SMT solvers (PCSat and Spacer) are tested. These tools are not designed for verifying the refinement $s \Rightarrow t$ we have tested so far, therefore, instead we perform verification by treating them as equivalence problems $s \equiv t$. Table 1 summarizes the verification results across benchmarks. Each column lists the verification outcome and the time required for each tool. For RECHECK, we also include the number of critical pairs generated for each problem.

Discussions. The first category (**Foundational calculi**) corresponds to the examples presented in §4.1-4.4. Since these examples go beyond the scope of functional programs, they are outside the range of other tools.

The next category (**Functional programs, our examples**) contains the program examples from §2 and §4. As seen in examples such as (**assoc**) and (**map/map**), the critical pair analysis in RECHECK can be regarded as an automation of induction over algebraic datatypes. This is similar to the inductionless induction and rewriting induction methods developed in rewriting theory [16, 27]. However, the crucial difference is that our method does not require termination. Proving termination of functional programs or calculi is often extremely difficult. Therefore, it is a major advantage over the rewriting-based induction methods (see also related work discussed in §4.4) for practical verification.

A key difference between RECHECK and the existing tools (MoCHi, RCaml, Timbuk) lies in their treatment of algebraic datatypes (ADTs). While the existing tools perform very well on properties over built-in integers, their SMT encodings make it difficult to reason about ADTs, such as Peano natural numbers or lists. For example, other tools could not verify a fairly standard equality problem on ADTs, such as (**assoc**), (**map/map**), (**map/append**). It is not clear why MoCHi reports unsafe for it. Likewise, it is not clear why (**map/map**) is reported as unsafe or unsat by other tools.

The example (**assocOP**) is the opposite direction of the associativity rule, taken as a refinement rule; in this case, RECHECK correctly reports that the evaluation steps do not decrease. The problem (**loop**) refines the non-terminating function defined by **loop** => **loop** into the constant 0. RECHECK can successfully verify such refinements for lazy, non-terminating programs as well.

The last category (**Functional Programs, existing problems**) consists of examples taken from other tools and variants. The problems (**fliter-exist**) and **even** are taken from Timbuk’s benchmark. Here, a limitation of RECHECK becomes apparent – its inability to handle integer problems. For instance, (**McCarthy 91**), the well-known McCarthy’s 91 function, which always returns 91, can be verified by SMT-based tools such as MoCHi and RCaml, but RECHECK cannot verify it. Even in the simpler **copy-copy (int)** example, where **copy** copies an integer, RECHECK fails because it does not support built-in integers. However, this also reveals an interesting contrast with SMT-based verifiers. When the same function is rewritten over algebraic datatypes of natural numbers (**copy-copy (nat)**), all other tools except for RCaml (PCSat) fail to verify it, whereas RECHECK succeeds naturally, as expected for such sim-

ple algebraic definitions. This experiment illustrates how RECHECK’s rewriting-based design provides native support for ADTs, complementing solver-oriented approaches that excel in numeric reasoning.

As for execution time, RECHECK completes all verifications in roughly milliseconds, about two orders of magnitude faster than the other tools. This speed advantage is attributed to the fact that RECHECK does not rely on an SMT solver.

5 Summary and Limitations

Summary. RECHECK provides an automated tool for verifying contextual improvement across diverse evaluation strategies. RECHECK rethinks automated verification by treating operational semantics as user-definable specifications rather than fixed theories. This allows reasoning not only within a language but about the language itself – experimenting with evaluation strategies, algebraic datatypes, and other non-standard features in a unified setting. In this sense, RECHECK serves as a platform for semantic experimentation exploring operational variation.

Current limitations. RECHECK cannot verify improvement established by techniques different from induction, such as foldr/build fusion [8]. Moreover, when verification requires many intermediate lemmas, a single automatic critical pair analysis may not suffice. As shown in §4.4, the verification can then be completed by interactively refining the set of rules through manual additions. The interactive design of RECHECK within GHCi is thus well suited for such iterative refinement of verification rules.

RECHECK currently assumes deterministic and state-less evaluation rules; non-deterministic or probabilistic semantics remain future work. Evaluation strategies and refinement rules must be user-supplied, and the reasoning is purely symbolic without constraint solving, so arithmetic-intensive analyses lie outside its scope. These are deliberate design choices: RECHECK complements SMT solver-based systems by extending automated reasoning into the semantic domain rather than numeric reasoning.

Future directions. Planned extensions include supporting for effectful and non-deterministic semantics, and integration with SMT or proof-assistant back-ends. We also plan to provide an extensible library of TERS specifications for common functional calculi, establishing RECHECK as a shared infrastructure for designing and analysing rewriting and operational semantics.

Acknowledgments. We are grateful to Koko Muroya, Shotaro Karasaki and Tomoya Ichikawa for valuable discussions and collaboration on the foundational theory and applications of RECHECK. This research was partially supported by a grant from the Mitsubishi Foundation (Hamana), and by JSPS KAKENHI Grant Numbers 24K02919 (Hamana) and 24K14898 (Emoto).

No. Problem		RECHECKCritical pairs		MoChi	RCaML (PCSat)	RCaML (Spacer)	Timbuk
Fundational calculi							
1	Maybe monad	YES	5	not supported	not supported	not supported	not supported
2	λ_{ml*} -calculus	YES	3	not supported	not supported	not supported	not supported
3	Effect handlers	YES	10	not supported	not supported	not supported	not supported
4	Call-by-need λ -calculus	YES	2	not supported	not supported	not supported	not supported
Functional programs (our examples)							
5	map/map	YES	2	Unsafe	Unsat	Unsat	NT
		0.0014s		1.739s	0.351s	0.195s	
6	assoc	YES	2	Unsafe	NT	NT	NT
		0.0010s		0.944s			
7	assocOp	MAYBE	2	Unsafe	NT	NT	NT
		0.0011s		1.261s			
8	map/append	YES	2	Exception: Stack overflow	Unsat	Unsat	NT
		0.0018s			1.016s	0.288s	
9	lenPlus	YES	2	Exception: Stack overflow	NT	SegFault	NT
		0.0012s				45.16s	
10	conj1	MAYBE	2	not supported	Sat	Sat	NT
		0.0010s			0.19s	0.197s	
11	conj2	YES	4				
		0.0016s					
12	ones	YES	1	Safe	Sat	Sat	NT
		0.0004s		0.332s	0.063s	0.073s	
13	refine	YES	1	Safe	Sat	Sat	ERROR
		0.0002s		0.386s	0.037s	0.046s	0.094s
14	revrev1	MAYBE	2	Unsafe	NT	OutOfMem	NT
		0.0008s		1.242s		68.036s	
15	revrev2	YES	4				
		0.0023s					
Functional programs (existing benchmark problems and variants)							
16	copy-copy (int)	not supported		Safe	Sat	NT	
				0.446s	0.823s		
17	copy-copy (nat)	YES	2	not supported	Sat	NT	NT
		0.0005s			0.185s		
18	makelist (int)	not supported		Safe	NT	NT	
				0.938s			
19	makelist (nat)	YES	1	not supported	NT	NT	NT
		0.0004s					
20	filter-exists	MAYBE	2	Unsafe	Unsat	Unsat	Proved
		0.0034s		1.193s	0.291s	0.193s	1.163s
21	even	YES	3	failure	Unsat	Unsat	Proved
		0.0006s			10.624s	0.1s	0.343s
22	McCarty91	not supported		Safe	Sat	Sat	NT
				0.619s	0.915s	0.083s	

Yes... Contextual improvement, MAYBE... Unknown, Sat... Satisfiable, Unsat... Unsatisfiable, NT... Non-Termination, Blank... Not tested

Table 1. Experiments

Data Availability Statement

The artifact accompanying this paper, which includes RECHECK, is available at <https://zenodo.org/records/17491532>. This archive contains all data and scripts necessary to reproduce the experiments.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
2. Faggian, C., Guerrieri, G., Treglia, R.: Evaluation in the computational calculus is non-confluent. In: 10th International Workshop of Confluence, IWC 2021. pp. 31–36 (2021)
3. Felleisen, M., Friedman, D.P., Kohlbecker, E.E., Duba, B.F.: A syntactic theory of sequential control. *Theor. Comput. Sci.* **52**, 205–237 (1987)
4. Fiore, M.: Second-order and dependently-sorted abstract syntax. In: Proc. of LICS’08. pp. 57–68 (2008)
5. Fiore, M., Hamana, M.: Multiversal polymorphic algebraic theories: Syntax, semantics, translations, and equational logic. In: 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013. pp. 520–529 (2013)
6. Fiore, M., Mahmoud, O.: Second-order algebraic theories. In: Proc. of MFCS’10. pp. 368–380. LNCS 6281 (2010)
7. Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding. In: Proc. of LICS’99. pp. 193–202 (1999)
8. Gill, A., Launchbury, J., Peyton Jones, S.L.: A short cut to deforestation. In: Proceedings of the conference on Functional programming languages and computer architecture. pp. 223–232 (1993)
9. Goguen, J.A., Meseguer, J.: Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theor. Comput. Sci.* **105**(2), 217–273 (1992). [https://doi.org/10.1016/0304-3975\(92\)90302-V](https://doi.org/10.1016/0304-3975(92)90302-V), [https://doi.org/10.1016/0304-3975\(92\)90302-V](https://doi.org/10.1016/0304-3975(92)90302-V)
10. Hamana, M.: Free Σ -monoids: A higher-order syntax with metavariables. In: Proc. of APLAS’04. pp. 348–363. LNCS 3302 (2004)
11. Hamana, M.: Polymorphic abstract syntax via Grothendieck construction. In: FoS-SaCS’11. pp. 381–395. LNCS3467 (2011)
12. Hamana, M.: A functional implementation of function-as-constructor higher-order unification. In: Proc. of International Workshop on Unification 2017. pp. 10–15 (2017), co-located with FSCD’17, Oxford, UK
13. Hamana, M.: How to prove decidability of equational theories with second-order computation analyser SOL. *Journal of Functional Programming* **29**(e20) (2019)
14. Hamana, M., Abe, T., Kikuchi, K.: Polymorphic computation systems: Theory and practice of confluence with call-by-value. *Sci. Comput. Program.* **187**, 102322 (2020). <https://doi.org/10.1016/J.SCICO.2019.102322>, <https://doi.org/10.1016/j.scico.2019.102322>
15. Haudebourg, T., Genet, T., Jensen, T.P.: Regular language type inference with term rewriting. *PACMPL* **4**(ICFP), 112:1–112:29 (2020)
16. Huet, G., Hullot, J.M.: Proofs by induction in equational theories with constructors. *Journal of computer and system sciences* **25**(2), 239–266 (1982)
17. Huet, G.P.: A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.* **1**(1), 27–57 (1975)

18. Kikuchi, K., Aoto, T., Sasano, I.: Inductive theorem proving in non-terminating rewriting systems and its application to program transformation. In: Proc. of PPDP 2019. pp. 13:1–13:14. ACM (2019)
19. Knuth, D., Bendix, P.: Simple word problems in universal algebras. In: Computational Problem in abstract algebra, pp. 263–297. Pergamon Press, Oxford (1970)
20. Maraist, J., Odersky, M., Turner, D.N., Wadler, P.: Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theor. Comp. Sci.* **228**(1-2), 175–210 (1999). [https://doi.org/10.1016/S0304-3975\(98\)00358-2](https://doi.org/10.1016/S0304-3975(98)00358-2)
21. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation* **1**(4), 497–536 (1991)
22. Moggi, E.: Notions of computation and monads. *Information and Computation* **93**, 55–92 (1991)
23. Morris Jr, J.H.: Lambda-calculus models of programming languages. Ph.D. thesis, Massachusetts Institute of Technology (1969)
24. Muroya, K., Hamana, M.: Typed term evaluation systems with refinements for contextual improvement, submitted for publication
25. Muroya, K., Hamana, M.: Term evaluation systems with refinements: First-order, second-order, and contextual improvement. In: Proc. of 17th International Symposium on Functional and Logic Programming (FLOPS’24). pp. 31–61. LNCS 14659 (2024)
26. Pretnar, M.: An introduction to algebraic effects and handlers. invited tutorial paper. In: Ghica, D.R. (ed.) The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015. Electronic Notes in Theoretical Computer Science, vol. 319, pp. 19–35. Elsevier (2015). <https://doi.org/10.1016/J.ENTCS.2015.12.003>, <https://doi.org/10.1016/j.entcs.2015.12.003>
27. Reddy, U.S.: Term rewriting induction. In: International Conference on Automated Deduction. pp. 162–177. Springer (1990)
28. Sabry, A., Wadler, P.: A reflection on call-by-value. In: Harper, R., Wexelblat, R.L. (eds.) Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24–26, 1996. pp. 13–24. ACM (1996). <https://doi.org/10.1145/232627.232631>, <https://doi.org/10.1145/232627.232631>
29. Sands, D.: Total correctness by local improvement in the transformation of functional programs. *ACM Trans. Program. Lang. Syst.* **18**(2), 175–234 (1996). <https://doi.org/10.1145/227699.227716>
30. Sato, R., Unno, H., Kobayashi, N.: Mochi: Software model checker for a higher-order functional language (demo presentation). In: 2012 ACM SIGPLAN Workshop on ML (ML 2012) (2012)
31. Sheard, T., Jones, S.P.: Template metaprogramming for Haskell. In: Proc. Haskell Workshop 2002 (2002)
32. Terese: Term Rewriting Systems. No. 55 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press (2003)
33. Unno, H., Kobayashi, N.: On-demand refinement of dependent types. In: Garigue, J., Hermenegildo, M.V. (eds.) Functional and Logic Programming, 9th International Symposium, FLOPS 2008, Ise, Japan, April 14–16, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4989, pp. 81–96. Springer (2008)
34. Unno, H., Terauchi, T., Kobayashi, N.: Automating relatively complete verification of higher-order functional programs. In: Giacobazzi, R., Cousot, R. (eds.)

- The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 75–86. ACM (2013). <https://doi.org/10.1145/2429069.2429081>, <https://doi.org/10.1145/2429069.2429081>
35. Unno, H., Torii, S., Sakamoto, H.: Automating induction for solving horn clauses. In: Majumdar, R., Kuncak, V. (eds.) *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Lecture Notes in Computer Science, vol. 10427, pp. 571–591. Springer (2017)
 36. Wadler, P.: Comprehending monads. In: *ACM Conference on Lisp and Functional Programming*. pp. 61–78. Nice, France (June 1990)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Data Structure Analysis for Binaries

Sadra Bayat Tork¹ , Nicholas
Coughlin^{1,2} , Alicia Michael^{1,2} , James
Tobler^{1,2} , and Kirsten Winter^{1,2}

¹ School of Information Technology and Electrical Engineering,
The University of Queensland, Australia

² Defence Science and Technology Group, Australia

k.winter,a.michael1,n.coughlin,james.tobler@uq.edu.au

Abstract. Data structure analysis (DSA) is a context- and field-sensitive points-to analysis developed for LLVM. It scales reasonably well to medium-size code with a sufficient precision to produce sound verification results. Our work is motivated by the difficulties encountered when analysing binaries. After compilation to machine-code level, binaries are lacking much of the structural information still present in LLVM code, rendering the core of the DSA approach, the unification of cells, insufficient to support precise verification.

To overcome this insufficiency, this work adapts the approach, thereby designing a *data structure analysis for binaries*, DSAbin. The paper proposes multiple options for the set up of our data structure analysis and compares their effects in an evaluation.

1 Introduction

Our work aims to support the analysis of binaries with the Boogie verifier [5] which enables us to provide assurance of memory safety and general information flow security. Analysing binaries instead of source code has multiple advantages, one of them being independence from compiler correctness which is difficult to guarantee [16]. It is also well known that standard compiler optimisations (e.g., dead-code elimination) may introduce security vulnerabilities that do not exist in the source code [10,26]. Working on (lifted) binaries brings the analysis much closer to the program that actually executes [4].³

Memory analysis is often a requirement for reasoning over, or soundly transforming, programs. It ranges over techniques such as value analysis, indirect call resolution, pointer analysis and alias analysis, which aim to over-approximate sets of memory locations and their values that are used in program operations.

Most approaches to binary analysis employ a coarse but scalable approximation to reconstruct the memory layout (e.g., [4,6,23,14]), often with the help of simplifying assumptions [25]. For a precise analysis that provides assurance

³ The trustworthiness of our binary analysis, of course hinges on the trustworthiness of our decompilation/lifting process, which lies outside the scope of this paper and has been discussed elsewhere [17,7].

of memory safety without too many false positives, however, these approaches alone are insufficient. While a precise representation of a flat memory leads to a blow-up of the verification task (disjointedness constraints of every pair of memory locations become unmanageable for the underlying solver), a coarse grouping into (potentially) aliasing regions leads to imprecise verification results.

Data Structure Analysis (DSA) [18,13,15] is a context- and field-sensitive memory analysis originally designed for LLVM IR. This analysis operates over nodes, potentially partitioned into cells, where each node represents a set of memory regions and their disjoint sub-regions. A points-to relation is represented as a graph over cells, such that each cell can point to at most *one* other cell. This creates the need to combine (i.e., merge or unify) multiple pointee cells into one, a process referred to as *unification*.

Lattner [18] constructs DSA out of three phases: a local, a bottom up (BU) and a top down (TD) phase. The local phase is performed intra-procedurally and handles both creating and unifying nodes from which it builds a *local graph*. The two latter phases (BU and TD) use the local graphs to render the approach inter-procedural by transferring information across function boundaries. At each call-site, relevant parts of the callee are cloned into each caller (BU) and vice versa from caller to callee (TD), thereby resolving calling constraints between actual and formal parameters. After these phases, all graphs include the complete context of their callees and callers, resulting in context sensitivity.

In this work, we focus on memory analysis in the context of Basil [21], a tool which generates semantically equivalent Boogie code [5] from ARM64 [1] binaries. The tool favours precision over scalability (as much as possible) with a focus on medium-sized binaries identified as safety- or security-critical components. Above all, this requires us to identify variable-like entities in the binaries to recover some precision of the semantics of memory accesses. Additionally, an alias analysis is necessary to drive analyses that lift the binaries to a level that is amenable to the verifier (e.g., an escape analysis that can provide information about what is accessed by a procedure and what is not). We have developed DSABin, a data structure analysis that is applicable to lifted binaries. While DSABin is generally applicable to any intermediate representation of binaries, we use it to support Basil and its intermediate representation, Basil IR.

Lattner’s approach as well as Gurfinkel et al.’s [13,15] approach utilise LLVM IR’s higher-level constructs such as variable entities and known types for those entities. Basil IR operates instead over addresses in a flat memory space. No notion of disjoint variables nor their types is maintained through the compilation. Lack of distinct memory entities inevitably leads to a coarser notion of memory objects. Local variables that are distinguished by LLVM IR are all represented by the same stack object in Basil. As a result, local variables become indistinguishable as soon as one local access cannot be identified and has to be over-approximated. Lack of type information on memory entities disallows us to use such information to reduce the number of memory constraints between operands with mismatched types as has been proposed by [18]. Applying the unification scheme of the standard DSA directly to a Basil IR program almost al-

ways leads to complete loss of field-sensitivity. Furthermore, type information on memory regions cannot be soundly recovered beforehand since any pre-analysis attempting to recover types would be incomplete without sound alias information.

To perform DSA despite these challenges, we introduce a framework that separates unification constraint *generation* from DSA-style constraint *processing*. Our constraint generation is supported by an intra-procedural *symbolic* value set analysis. Postponing the merging of nodes and cells to after the value set analysis has completed (rather than conducting both in lockstep) is vital for the analysis of binaries where a memory access usually does not reveal which address is targeted prior to the value set analysis (leading to early merging that cannot be undone). Delaying the merging is thus beneficial. The separation also supports an easy instrumentation of options for various precision levels of the analysis by tailoring the abstract domain of symbolic values and the constraint generation.

Analysing binaries also requires us to handle the lack of type information. As a solution, we propose a novel approach to the unification of cells which only relies on size information provided through load and store instructions.

Contributions. We propose a novel approach to memory region analysis for lifted binaries based on data structure analysis to provide precision that surpasses existing approaches for binaries. In our work, we combine the traditional DSA approach with an intra-procedural *symbolic value set analysis* which produces memory access constraints. Additionally, we adjust the approach to binary code using a *novel unification scheme*, which is capable of soundly relying on memory access sizes. Our evaluations showcase the effect of these improvements on precision and performance.

Although the disassembly of binaries and the problem of control flow reconstruction is not further discussed here, we find that DSABin helps to resolve indirect calls which the disassembler fails to resolve. Additionally, Basil’s DSABin generates assertions on memory safety where a memory access is symbolic. The Boogie back-end of our framework enables checking the validity of such assertions, thereby providing strong guarantees on memory safety.

2 Data Structure Analysis in Basil: an overview

The primary purpose of our work is to identify aliasing relationships between fields of memory. For binaries, this requires the reconstruction of the memory layout, and the data structures stored in it (i.e., “variable-like” entities). We aim to associate each memory operation in the program with the possible memory regions it *may* operate over and, where possible, which field of the region. The memory regions that can be recognised are (i) the *stack* frame, corresponding to memory on the stack associated with a function, (ii) *heap* regions corresponding to memory returned by allocators such as malloc, and (iii) *global* memory, corresponding to the static, fixed set of global program variables. It is important to note that at the binary level the structure of these regions is unclear.

A *node* is representative of one or more data structures accessed in the program. It consists of distinct *cells* which represent fields in the data structure. Each cell points to at most one other cell in a graph that defines the points-to relation. To maintain the invariant of a functional points-to relation, cells and their nodes have to be *merged*, i.e., combined into one, when a cell points to multiple cells. This process is referred to as *unification*. Consequently, merged cells and nodes represent multiple fields and data structures. To approximate complex behaviour in the program, nodes may be *collapsed*, i.e., all their cells are fused into one, thereby losing field-sensitivity. Merging a node with a collapsed node results in loss of field-sensitivity of the resulting node.

Nodes and cells, and their points-to relation, are derived from flow-insensitive bidirectional *constraints* that summarise the memory accesses of the program. These constraints allow for an efficient unification-based points-to analysis [24]. Given their flow-insensitivity, the processing of symbolic constraints can be delayed until further information is known about the constraint arguments. As such, DSA can maintain *partial* graphs with only a subset of constraints processed while remaining sound. Such graphs are then completed as constraints external to the procedure are injected into the graph (e.g., in the case of indirect call resolution).

Symbolic values. We split the task of the procedure-local phase into a flow-sensitive *pre-local phase* and a subsequent *local phase*. The pre-local phase has two purposes: (i) it assigns *symbolic values* to each virtual, single assignment register [9,11] by (abstractly) evaluating program statements and (ii) it emits *unification constraints* between symbolic register values and memory accesses. Data structure nodes and cells are created from symbolic values for each region that is referenced. The local and the BU/TD phases, standard in DSA, then maintain points-to relations over the nodes by merging/unifying nodes and cells according to the generated constraints.

Unification. Unification is the basis for efficiently computing field-sensitive and context-sensitive points-to relations. It is the most important graph operation and the basis of other operations such as collapse and the transfer of the calling context between caller and callee. As in standard DSA and other unification-based approaches ([18,13,15]), our approach to unifying cells requires the unification of the nodes to which they belong, and subsequently the unification of all cells reachable via the points-to edge. A simple example can be seen in Fig. 1. After compilation of the code on the left, our encoding of DSA for binaries produces a simple graph, depicted on the top right. All globals are represented by a single node in this graph as it is not clear how they should be divided. In some cases, we can exploit information in the binary, such as the symbol table, as a heuristic for the structure of globals. Similarly, all local variables on the stack are represented by a single collapsed node.

We diverge from [18,13,15] in how we unify without irreversibly collapsing the resulting node. Whereas both Lattner [18] and Gurfinkel et al. [13,15] require stricter alignment of cells (and corresponding types) we enable merging of cells of different sizes and positions.

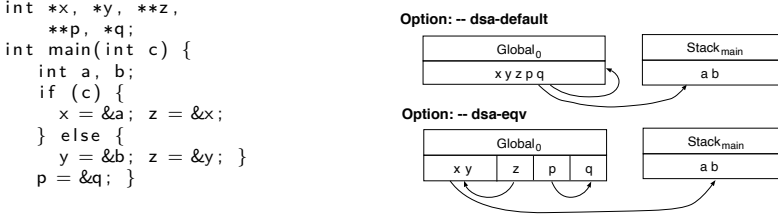


Fig. 1: Merging cells in a single node vs node collapse

Generally, finer granularity of cells in a node provides more structural and more precise points-to information. That is, a node with a single cell containing multiple offsets represents multiple memory spaces (starting at the offsets). A node with multiple cells, each containing a single offset, indicates that we can distinguish the memory spaces of each cell. The cells (and the memory they represent) are known to be separate, thereby supporting tracking of distinct points-to relationships for each.

Options. To balance the trade-off between scalability and effectiveness of the unification, we experiment with three different settings: (i) option *dsa-default* maintains only one region for global memory and does not allow the merging of cells within one node (ii) option *dsa-split* introduces a separate region for each global variable or structure (retrieved via symbol table entries), and (iii) option *dsa-eqv* enables the unification of cells within one node, avoiding collapse.

Fig. 1 illustrates the difference between option (i) and (iii). The graphs created from the binary of the given code represent all global variables as cells in a single global node, and the two local variables *a* and *b* as cells in the stack region. The default option results in a collapse of the (single) global node. Field-sensitivity cannot be maintained since *z*, represented by a cell in the global node, can point to both global variables *x* and *y* and merging the respective cells is prohibited. Under option *dsa-eqv* only the cells for *x* and *y* are merged, and field-sensitivity is partly maintained. Note that merging of *x* and *y* also leads to merging of cells *a* and *b* in the stack region.

Fig. 2 demonstrates the difference between options (i) and (ii). It shows the effect of recognising a single versus multiple global regions, one per global variable. Since the global *array* is accessed at an unknown index *r* (represented by node *Par_{main}* pointing into the array), the structure of the array is lost (its global node collapsed), whereas global variables *x* and *y* are maintained given they are represented in their own nodes, *Global₁₃₁₁₆₈* and *Global₁₃₁₁₇₆* respectively⁴. Under the default option, with a single node for all globals, the collapse of the array results in the collapse of the whole global region.

⁴ As indices for the global nodes we use the offset of the base address to the corresponding memory regions as these provide unique node identifiers. For the stack cells for which no local variable names are available in the C source, we use ranges of stack-relative offsets for identification.

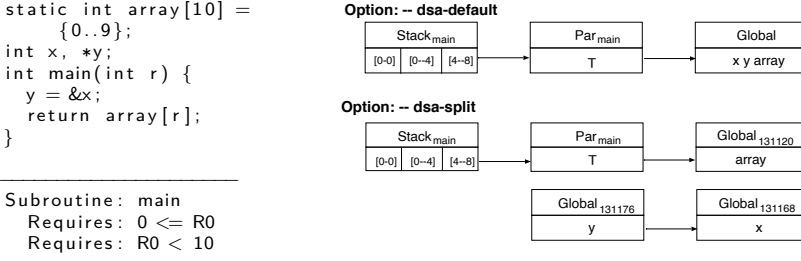


Fig. 2: Graph with a single or multiple global regions

Dividing the global memory is based on a heuristic and not necessarily sound, e.g., variables x and y might be accessed via large values of r . To ensure the assumptions made during *DSAbin* are sound, under option *dsa-assert* memory safety assertions are introduced into the program, thereby bounding accesses to the assumed memory regions. The verifier (*Boogie* [5] in our case) will check that the address of the memory operation falls within the bounds of the global.

For the program in Fig. 2, `assert(0 ≤ R0 < 10)` is inserted before the access to the array ($R0$ being the register holding r). The program will only verify with a specification that bounds $R0$ as shown at the bottom left of Fig. 2.

3 Pre-local phase

In low-level code most memory addresses are calculated via bit-vector expressions spread across a series of instructions. Various complex idioms are present, such as the partial construction of addresses and pointer tagging. These complex operations obfuscate the underlying memory behaviour, hampering the flow-insensitive analysis during DSA's local phase. The pre-local phase focuses on simplifying and reducing these idioms to simple constraints, based on typical memory access behaviours. Collecting possible values for registers allows us to track locations of memory accesses (loads and stores) and to support the generation of (flow-insensitive) constraints between memory content and their (potentially abstract) memory locations.

We phrase the computation of symbolic values as a flow-sensitive dataflow analysis over *Basil* IR in single assignment form [9,11], and refer to it as symbolic value set analysis. The statements of the IR, identifiable by line numbers, include register assignments, memory loads and stores (with access size), function calls and returns.

$$\begin{aligned}
 stmt ::= & R_L := e \mid R_L := mem(e): sz \mid mem(e): sz := R_R \mid \\
 & R_L := Call(f, R_0 \dots, R_n) \mid Return(R_R)
 \end{aligned}$$

We also consider procedure headers `procedure  $f, r, p_0, \dots, p_n$` with return parameter r and formal parameters p_i .

3.1 Symbolic Value Set Analysis

Our symbolic value analysis (SVA) tracks *symbolic values* for each virtual register. A symbolic value is pair of symbol (representing a memory region) and offset values (pointing into the memory region), $\mathcal{SV} \subseteq \mathcal{Rgs} \times \text{ValueSet}$. It allows us to capture memory accesses abstractly where they cannot be evaluated and are given in terms of unknown values (from loads, returns, or function parameters).

A *symbol*, $sb \in \mathcal{Rgs}$, may represent a procedure stack, Stack_{proc} , global memory, Global_i , or a heap region, Heap_i , each of which covers a certain address space. An integer value i known not to be an address is represented as $(\text{Int}, \{i\})$. We also introduce placeholder regions for values loaded from memory, Load_i , or input/output parameters of a procedure which are unknown during the pre-local phase, $\text{Par}_{(proc,i)}$ and Ret_{proc} , which we collectively refer to as *placeholders*. Using placeholders allows us to generate memory constraints without knowing which region is referenced.

A *ValueSet* is an abstraction of integer sets, with distinct values $\top$ and $\perp$, a meet $\sqcap$ and join operator $\sqcup$, and support for operations $\oplus^{vs}$. For illustrative purposes, we directly consider a set of integers with configurable widening for termination. Unknown value sets are over-approximated to “any value”, $\top$. For example, the symbolic value representing a set of local variables of procedure p at offsets off_1 and off_2 is represented as $(\text{Stack}_p, \{off_1, off_2\})$, a global variable at address g as $(\text{Global}_0, \{g\})$ if global memory is a single region, or as $(\text{Global}_g, \{0\})$ if the global region is split. A (non-address) integer i is represented as $(\text{Int}, \{i\})$. The symbolic value $(\text{Stack}_p, \top)$ represents an access anywhere on the stack of procedure p .

Abstract and concrete domain. We express SVA in the abstract interpretation framework [8] using the abstract value domain of symbolic values, $\mathcal{P}(\mathcal{SV})$. Join and widen operators in the abstract value domain are derived from the underlying ValueSet domain, applied point-wise per base symbol. We point-wise lift this abstract domain to the state abstract domain $\text{SVA}^\sharp$ which maps each register to a set of symbolic values, with abstract state $\sigma^\sharp \in \text{Reg} \rightarrow \mathcal{P}(\mathcal{SV})$.

Symbolic expressions are evaluated to symbolic values when they constitute a simple address calculation. To distinguish integers from globals, we assume a partial function gba mapping an address to the base of its containing global region such $\forall i, j \in \text{dom}(gba). gba(i) \leq i \wedge gba(i) \leq j \leq i \Rightarrow gba(i) = gba(j)$.⁵

Note that gba is relative to the option setting (see Sec. 2). Under the default option, the global address space, $\text{ran}(gba)$, is one flat structure, under *dsa-split*, it is informed by individual entries in the symbol table.

$\forall i \in \text{Int}$ and $\forall R \in \text{Reg}$,

$$\begin{aligned} \text{eval}_{sva}^\sharp(i) &= \begin{cases} \{(\text{Int}, \{i\})\} & \text{if } i \notin \text{dom}(gba) \\ \{(\text{Global}_{gba(i)}, \{i - gba(i)\})\} & \text{otherwise} \end{cases} \\ \text{eval}_{sva}^\sharp(R) &= \sigma^\sharp(R) \end{aligned}$$

Over expressions, we define $\text{eval}_{sva}^\sharp(e_1 \oplus e_2) = \text{eval}_{sva}^\sharp(e_1) \oplus^{sva} \text{eval}_{sva}^\sharp(e_2)$, where $\oplus$ denotes some binary operator in **Basil IR** which abstracts to an op-

⁵ We refer to the domain and the range of a function using dom and ran , respectively.

$$\begin{array}{c}
\text{procedure } f, r, p_0, \dots, p_n \\
\text{fresh}(\text{Stack}_f) \quad \forall 0 \leq k \leq n. \text{fresh}(\text{Par}_{(f,k)}) \quad \text{fresh}(\text{Ret}_f) \\
\text{[Entry]} \frac{}{\sigma^\# := \sigma^\# [R31 \mapsto (\text{Stack}_f, \{0\}), \forall 0 \leq k \leq n. R_k \mapsto (\text{Par}_{(f,k)}, \{0\})]} \\
\\
\text{[Local]} \frac{R_L := e}{\sigma^\# := \sigma^\# [R_L \mapsto \text{eval}_{sva}^\#(e)]} \quad \text{[Load]} \frac{R_L := \text{mem}(e): sz \quad pc = i \quad \text{fresh}(\text{Load}_i)}{\sigma^\# := \sigma^\# [R_L \mapsto (\text{Load}_i, \{0\})]} \\
\\
\text{[Malloc]} \frac{R_L := \text{Call}(\text{malloc}, R_1) \quad pc = i \quad (Int, sz) \in \sigma^\#(R_1) \quad \text{fresh}(\text{Heap}_i)}{\sigma^\# := \sigma^\# [R_L \mapsto (\text{Heap}_i, \{0\})]} \\
\\
\text{[Call]} \frac{R_L := \text{Call}(f, R_0 \dots, R_n) \quad pc = i \quad \text{fresh}(\text{Ret}_i)}{\sigma^\# := \sigma^\# [R_L \mapsto (\text{Ret}_i, \{0\})]}
\end{array}$$

Fig. 3: Transfer function for the symbolic value analysis

erator $\oplus^{vs}$ over the value set domain. If both operands are integers a standard evaluation over value sets simplifies the expression. If one operand is a (non-address) integer and the operator is either addition or subtraction, $\oplus \in \{+, -\}$, we have $(sb_1, vs_1) \oplus^{sva} (Int, vs_2) = (sb_1, (vs_i \oplus^{vs} vs_j))$. All other expressions result in an over-approximation with value set $\top$, $\{(sb_1, \top), (sb_2, \top)\}$, indicating that regions sb_1 or sb_2 may be accessed at some (unknown) offset into the region.

The corresponding concrete value domain is the set of 64-bit bitvectors, corresponding to the architecture's memory address width. As we consider binary programs, there is no distinction between integer values and addresses. Consequently, this set captures both. The concretisation function for the symbolic value domain, γ_{SV} is relative to the base addresses of regions $\rho : \mathcal{Rgs} \rightarrow \mathcal{P}(Int)$ (e.g., $\rho(\text{Global}_i) = \{i\}$), which is invariant over the execution of the program. With $+$ the point-wise addition we have the following definition.

$$\gamma_{SV}((sb, vs), \rho) = \{a \mid \exists v \in vs. a = \rho(sb) + v\}$$

Transfer Function. The symbolic value analysis computes the abstract state over the instructions using the transfer function of Figure 3. For each initial access to a region we additionally assume a *fresh* symbol as part of the domain $\mathcal{Rgs}$, implicitly requesting that it symbolises a region disjunct from others.

Upon **entry** to a procedure f which has return parameter r , and $n+1$ function parameters $p_0, \dots, p_n$, the abstract state $\sigma^\#$ is updated with a symbolic value for the stack pointer R31, as well as symbolic values for the registers holding the formal parameters of f . Each of these symbolic values has a *nil* offset (i.e., the value set is $\{0\}$). We assume fresh symbols for the stack, and the formal input and return parameters, the latter being required for the generation of constraints (Sec. 3.2).

For **local updates** the abstract state $\sigma^\sharp$ is updated for the left-hand-side register R_L by the symbolic value of the right-hand-side expression. Where the expression refers to a placeholder, its full evaluation is delayed until the placeholder can be resolved (e.g., during the local, BU or TD phase). For a **load** instruction at line i accessing memory via $mem(e)$, Rule [Load] assumes a fresh symbol $Load_i$ and assigns its corresponding symbolic value to register R_L . For a **heap allocation** at line i , we assume a fresh symbol for the heap, $(Heap_i)$, and update $\sigma^\sharp$ with a new value for register R_L . A general **call** instruction assumes a fresh placeholder⁶ for the return value of the callee, and updates the state of register R_L to the unknown symbolic result value. **Store** instructions do not affect the symbolic values in $\sigma^\sharp$.

3.2 Constraint Generation

We generate constraints over symbolic values to represent the linking of memory locations (or their contents) which may be aliased (i.e., pointing to the same memory locations as other memory locations). The constraint relation $\Leftrightarrow$ is symmetric and as such un-directed [24]. A constraint causes the merging of all involved memory locations, which is the task of the local analysis (Sec. 4.2).

We denote dereferencing of a symbolic value as $\llbracket \cdot \rrbracket_s$, which refers to the *content* of a memory location, represented by a symbolic value, of size s . $\llbracket \cdot \rrbracket_s$ refines during the local phase to *points-to* edges over cells in the data structure graph, and additionally allows us to determine the size of a cell from the memory access. As a shorthand we omit s to denote single byte access. Constraints link memory locations to either other memory locations $sv_i \Leftrightarrow sv_j$, or with the content of another memory location $\llbracket sv_i \rrbracket_s \Leftrightarrow sv_j$. The constraint relation also extends to sequences of symbolic values, $\langle sv_0, \dots, sv_n \rangle \Leftrightarrow \langle sv'_0, \dots, sv'_n \rangle$, to express the link between actual and formal parameters.

To inform later phases of the DSA, constraints can be labelled with the current function context and the calling context if they are between a caller and a callee, i.e., $\Leftrightarrow_{(f,i)}^g$ is generated for a call from caller f at call site i to callee g . A constraint $\Leftrightarrow_{(f,i)}^g$ between two symbolic values will cause the BU/TD phases to clone the nodes of the formal parameters of the callee g into the caller f and subsequently unify them with the nodes of the actual parameters at call site i .

Transfer Function. Constraints are generated only for memory operations and call/return instructions (Fig. 4), as these affect the shape of the data structure graph and its points-to relation. They are relative to the current procedure *proc*.

A **load** instruction creates a constraint between the abstract state $\sigma^\sharp$ of the register R_L on the left with the *content* of the memory accessed at the (abstract) address $eval_{sva}^\sharp(e)$. Effectively, the constraint will be between $mem(e)$'s content

⁶ Note that fresh variables are statically produced per program statement (on entry and for load and call instructions), and not per execution of a statement. Since the programs have finitely many statements only finite variables are generated.

$$\begin{array}{c}
\text{[Load]} \frac{R_L := \text{mem}(e) : sz \quad \text{proc} = f}{\llbracket \text{eval}_{sva}^\#(e) \rrbracket_{sz} \Leftrightarrow_f \sigma^\#(R_L)} \quad \text{[Store]} \frac{\text{mem}(e) : sz := R_R \quad \text{proc} = f}{\llbracket \text{eval}_{sva}^\#(e) \rrbracket_{sz} \Leftrightarrow_f \sigma^\#(R_R)} \\
\\
\text{[Call]} \frac{R_L := \text{Call}(g, R_0 \dots, R_n) \quad pc = i \quad \text{proc} = f}{\langle \sigma^\#(R_0), \dots, \sigma^\#(R_n) \rangle \Leftrightarrow_{(f,i)}^g \langle \text{Par}_{(g,0)}, \dots, \text{Par}_{(g,n)} \rangle} \\
\quad \sigma^\#(R_L) \Leftrightarrow_{(f,i)}^g (\text{Ret}_g, 0) \\
\\
\text{[Return]} \frac{\text{Return}(R_R) \quad \text{proc} = f}{(\text{Ret}_f, 0) \Leftrightarrow_f \sigma^\#(R_R)}
\end{array}$$

Fig. 4: Transfer Function

and a (collection of) fresh symbolic value(s) Load_i created by the previous SVA analysis (Rule [Load], Fig. 3). A **store** instruction emits a constraint that links the *content* of the accessed memory with the symbolic value of the register on the right. This effectively creates a points-to relation from the memory location on the left to the symbolic value of the register R_R . A **call** instruction generates a set of constraints over all parameter placeholders of the callee g linking them with the abstract state of the actual parameters $\sigma^\#(R_i)$. Additionally, the rule emits a constraint that links the callee's return value Ret_g (generated by Rule [Entry], Fig. 3), with the result register (R_L) at this call-site in the caller f . Note that the generated constraint is call-site specific, providing context-sensitivity. The transfer function for the **return** instruction emits a constraint between the procedures' formal return parameter Ret_f (established via Rule [Entry], Fig. 3) and the register containing the returned value, $\sigma^\#(R_R)$. This approach can easily be extended to multiple return parameters.

4 Forming local DSAbin graphs

The local phase creates a *procedure-local* data structure graph (DSG) over cells, by processing each procedure's local memory constraints generated in the pre-local phase. Cells in the graph are grouped into nodes.

The initial set of nodes and cells in the local DSG is generated by evaluating symbolic values (sb, vs) collated in the pre-local analysis: Each symbol sb generates a node (which is abstract in the case sb refers to a placeholder region), and each value in the value set $v \in vs$ provides the offset to a cell within the node. The content mapping $\llbracket \cdot \rrbracket_s$ in the generated constraints gives rise to edges in the local DSG as well as the size of cells. The graph created from these elements requires merging or *unifying* cells to maintain the **invariant** over edges: each cell has only one pointee. After unification, each procedure's DSG represents a sound approximation of the local memory access patterns.

4.1 Data Structure Graphs

Nodes. Each symbol identified in the SVA that is not *Int* (whether a placeholder or not) is initially converted to a distinct node. Further processing of the constraints generated in the pre-local phase leads to the generation of new nodes (defined in Sec. 4.2) which will represent a set of memory regions. Each memory region can belong to at most one node *after* this process.

Placeholder nodes are created from symbolic values with a placeholder symbol, e.g., when a memory load constraint is processed prior to processing any memory write constraints involving the same memory address. A placeholder node represents an abstract pointee (loaded value). Once a corresponding memory store constraint to the same address is processed, the placeholder node can be unified with the node representing the region accessed by the store instruction.

Cells. Cells of a node are created from symbolic values (*sb*, *vs*), such that the node represents the region matching the symbol *sb*, and each cell represents a *node-relative* offset from the value set *vs*. As such cells represent disjoint program variables or structures which always have to be accessed disjointly. Each variable or structure occupies memory with addresses ranging over a node-relative interval. For example a scalar variable is addressable by a single address, an array over integers of length four, when viewed as a whole, encompasses four offsets into the structure. Cells are grouped into nodes which represent the memory regions they belong to. This allows us to distinguish variables on the heap from those on the stack without knowing the actual value of the base address of these regions. Accordingly, we represent a cell as a set of integers which, together with the node's base address, correspond to memory addresses.

Definition 1. A cell is a set of positions in a node, $c \in (N \times \text{ValueSet})$, with *c.n* being *c*'s node, and *c.s* its *ValueSet*. *Cells* denotes the set of cells.

When a memory access spans multiple bytes, the created cell covers an offset range of that size. To adjust the size of a cell, we refer to the size of a memory access indicated by the load and store instructions.

A cell could also have unknown size if it belongs to a node which has lost field-sensitivity. Collapsed nodes with a single cell are introduced when a memory region is accessed at a statically unknown offset. Since a cell in a collapsed node overlaps with all other cells in the same region, a single occurrence of $\top$ is enough to lose all field information for the region.

Unification of cells is based on the *merge* operator, $\sqcup^*$, which combines cells into one when they *overlap*. We define both notions over the abstract domain *ValueSet* (cf. Sec. 3.1). Two sets of values overlap if their intersection, i.e., the meet $\sqcap$ of the abstract domain, is not empty, i.e., $\text{overlap}(s_1, s_2) = s_1 \sqcap s_2 \neq \perp$.

To merge the cells in *C* with the same node, i.e., $\exists n. \forall c \in C. c.n = n$, all overlapping value sets are joined while others remain. Since merged pairs can overlap with each other, we compute the fixpoint over the set of *ValueSets* *S*.

$$\begin{aligned} \sqcup^* (S) = \mu S. (\{ & s_1 \sqcup s_2 \mid s_1, s_2 \in S \wedge \text{overlap}(s_1, s_2) \}) \\ & \cup \{ s \mid s \in S \wedge \forall s' \in S. \neg \text{overlap}(s, s') \} \end{aligned} \quad (1)$$

Graphs. A Data Structure Graph (DSG) represents a points-to relation between memory-regions.

Definition 2. A Data Structure Graph (DSG) is given as a tuple (C, E) with $C: \mathcal{P}(\text{Cell})$ a set of cells, and $E: C \times C$ a relation between cells.

From constraints to graphs. Let $Cstr$ be the set of constraints collected by the constraint generation. Dereferencing a symbolic value symbolises a points-to edge from the cell containing the address to the pointee cell assigned to R . We define the set of points-to edges E generated from the constraints and cells C as follows. The set of nodes is derived as $N = \{n \mid \exists c \in C. c = (n, -)\}$.

$$\begin{aligned} E &= \{((sb, \sqcup \{vs..vs +^\# sz\}), c') \mid \exists e, sz, R. \\ &\quad \llbracket eval_{sva}^\#(e) \rrbracket_{sz} \Leftrightarrow \sigma^\#(R) \in Cstr \wedge (sb, vs) \in eval_{sva}^\#(e) \wedge c' \in \sigma^\#(R)\} \\ C &= \{c \mid (c, -) \in E \vee (-, c) \in E\} \end{aligned}$$

Definition 3. A data structure graph (C, E) with nodes N is wellformed if it satisfies two invariants over the structure.

Inv 1. Cells in each node are disjoint,

$$\forall n \in N, c, c' \in C. c.n = c'.n \Rightarrow \neg \text{overlap}(c.s, c'.s)$$

Inv 2. The edge relation E is a function over cells,

$$\forall c_1, c_2, c_3. (c_1, c_2) \in E \wedge (c_1, c_3) \in E \Rightarrow c_2 = c_3.$$

To establish invariant **Inv 1**, we must for each node, merge the cells that are overlapping. Let $(C^{\sqcup^*}, E^{\sqcup^*})$ be the graph generated such that overlapping cells in the same node are merged, given $c \leq c' = (c.n = c'.n \wedge c.s \subseteq c'.s)$.

$$\begin{aligned} C^{\sqcup^*} &= \{(n, s) \mid s \in \sqcup^*(\{c.s \mid c.n = n \wedge c \in C\}) \wedge n \in N\} \\ E^{\sqcup^*} &= \{(c_1, c_2) \mid (c_3, c_4) \in E \wedge c_3 \leq c_1 \wedge c_4 \leq c_2 \wedge \{c_1, c_2\} \subseteq C^{\sqcup^*}\} \end{aligned}$$

To create a *wellformed* DSG, and thereby a solution to the data structure analysis, we need to ensure invariant **Inv 2** holds while maintaining **Inv 1**. This requires the unification of cells and nodes along the points-to edges. A wellformed data structure graph G is then defined as

$$G = \text{unifyCells}(\text{ptees}(C^{\sqcup^*}, E^{\sqcup^*}), C^{\sqcup^*}, E^{\sqcup^*})$$

where the sets of pointees $\text{ptees}(C^{\sqcup^*}, E^{\sqcup^*})$ and the unification operation unifyCells are defined in the following section.

4.2 Unification

One of the two main contributions of **DSAbin** is the adaptation of the unification of cells and nodes for binary analysis. We tailor the conditions under which cells can be merged and how to adjust them to achieve an alignment. Like in other works on points-to analyses, our algorithm is encoded as a highly efficient

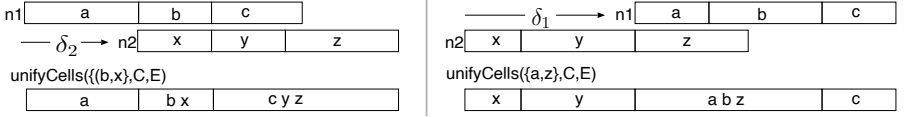


Fig. 5: Unification of cells b and x (left) and a and z (right) in a graph (C, E)

union-find algorithm [24]. When unifying cells, we first identify the smallest offset implied by the cells per node. The distances between these smallest offsets serve as the shifts for alignment. That is, the unification of cells is defined as the unification of cells of offset-adjusted nodes.

Example. Fig. 5 shows the effect of unification of two cells with mismatched cells at various offsets. In the case where cells b (at offset 2 in node $n1$) and x (at offset 0 in node $n2$) are unified, all cells of node $n2$ are shifted by $\delta_2 = 2$. Now cells b and x are aligned, resulting in cells c, y and z overlapping and consequently being merged into one new cell; cell a remains. Unifying cells a (at offset 0 in node $n1$) and z (at offset 3 in node $n2$) on the right requires shifting $n1$ by $\delta_1 = 3$; cells a and z are aligned but also overlap with cell b which needs to be merged together with a and z .

Let uc be the set of cells to be unified, $\mathcal{Nodes}(uc)$ the set of nodes containing uc , and c_n^{uc} the set of cells to be unified that belong to node n .

$$\begin{aligned}\mathcal{Nodes}(uc) &= \{c.n \mid c \in uc\} \\ c_n^{uc} &= \{c \mid c \in uc \wedge c.n = n\}\end{aligned}$$

To unify uc , we first combine cells that share a common node n into one cell by approximating their positions in n . We refer to the resulting set of cells as $\text{joinCells}(uc)$. The set of cells involved in the unification, $\text{ufyCells}(uc)$, includes these and any contained by nodes in $\mathcal{Nodes}(uc)$ but not uc explicitly.

$$\begin{aligned}\text{joinCells}(uc) &= \{(n, \tilde{\bigcup}_{c \in c_n^{uc}} c.s) \mid n \in \mathcal{Nodes}(uc)\} \\ \text{ufyCells}(uc) &= \{c \mid c.n \in \mathcal{Nodes}(uc) \wedge c \notin uc\} \cup \text{joinCells}(uc)\end{aligned}$$

where $\tilde{\bigcup} s$ equals $\bigcup s$ if *dsa-eqv* is enabled but $\top$ for a non-singleton s otherwise.

The alignment process aims to adjust cells in $\text{ufyCells}(uc)$ such that the cells in $\text{joinCells}(uc)$ start at a common position. This adjustment must be uniform per node to preserve node relative positions. Consequently, an offset per node $n \in \mathcal{Nodes}(uc)$ is determined. The smallest offset of a cell c is given as $s_{\min}(c) = s_i$ such that $\forall s_j \in c.s, s_i \leq s_j$. Therefore, the starting offset $\text{start}_{uc}(n)$ for a node n is given by offset $s_{\min}(c)$ of its respective cell c in $\text{joinCells}(uc)$. The maximal of these starting offsets, startMax_{uc} , serves as the common starting position, with all nodes offsetting by some factor $\delta_{uc}(n)$ to ensure the cells $\text{joinCells}(uc)$ start at this point.

$$\begin{aligned}\text{start}_{uc}(n) &= s_{\min}(c) \text{ given } c \in \text{joinCells}(uc) \wedge c.n = n \\ \text{startMax}_{uc} &= \max\{\text{start}_{uc}(n) \mid n \in \mathcal{Nodes}(uc)\} \\ \delta_{uc}(n) &= \text{startMax}_{uc} - \text{start}_{uc}(n)\end{aligned}$$

The set of adjusted cell positions $adjCells(uc)$ is then defined as

$$adjCells(uc) = \{c.s +^\# \delta_{uc}(c.n) \mid c \in ufyCells(uc)\}$$

where the operator $+^\#$ captures the shifting of each offset in a set by a given amount, i.e., for $s = \{v_1..v_2\}$ we define $s +^\# \delta = \{v_1 + \delta..v_2 + \delta\}$.

Example. In Fig. 5 we have $startMax_{\{b,x\}} = 2$ with $s_{min}(b) = 2$, $s_{min}(x) = 0$ and hence $\delta_{\{b,x\}}(n_1) = 0$ and $\delta_{\{b,x\}}(n_2) = 2$, hence $adjCells(\{b,x\}) = \{a, b, c, x +^\# 2, y +^\# 2, z +^\# 2\}$ such that all cells in node n_2 are shifted along. Note that $adjCells(\{b,x\}) \neq adjCells(\{a,z\})$ since $\delta_{\{a,z\}}(n_1) \neq \delta_{\{b,x\}}(n_1)$.

We construct a *fresh node*, n_{new} , to represent the unification of cells in uc . The new cells of this new node, N^{uc} , are derived by merging the adjusted position sets $adjCells(uc)$ to preserve **Inv 1**.

$$N^{uc} \triangleq \{(n_{new}, s) \mid s \in \sqcup^* adjCells(uc)\}$$

With the new node constructed, the graph structure (C, E) must be updated to reflect the cell replacement, producing a new graph (C^{uc}, E^{uc}) . For instance, edges that previously referenced cells in nodes $Nodes(uc)$ must now reference corresponding cells in N^{uc} . We achieve this by defining a map function that redirects each cell involved in the unification to its corresponding cell in N^{uc} , while leaving all other cells unchanged.

$$map^{uc}(c) \triangleq \begin{cases} c' & \text{where } c.n \in Nodes(uc) \wedge c' \in N^{uc} \wedge c.s +^\# \delta_{uc}(c.n) \subseteq c'.s \\ c & \text{otherwise} \end{cases}$$

Applying this map to all cells and edges yields the updated graph.

$$\begin{aligned} C^{uc} &\triangleq \{map^{uc}(c) \mid c \in C\} \\ E^{uc} &\triangleq \{(map^{uc}(c), map^{uc}(c')) \mid (c, c') \in E\} \end{aligned}$$

This unification process can introduce new violations of the single points-to invariant. After updating the edges E^{uc} , a cell in N^{uc} may point to multiple distinct cells, requiring those cells to be unified in turn. To manage this effect, we employ a worklist $Uc = \{uc_1, \dots, uc_n\}$ of cell sets awaiting unification. After unifying a set uc , we identify new candidates by examining each cell in N^{uc} and collecting any non-singleton pointee sets given E^{uc} . We encode this as a function $ptees(R, E)$, taking a set of cells R to check and edges E .

$$ptees(R, E) \triangleq \{S \mid \exists c \in R. S = \{c' \mid (c, c') \in E\} \wedge |S| > 1\}$$

An updated worklist Uc^{uc} is derived by removing the processed set uc and adding the new candidates $ptees(N^{uc}, E^{uc})$. Additionally, as the graph has changed, the map function must be applied to all remaining worklist entries to ensure they reference current cells.

$$Uc^{uc} \triangleq \{\{map^{uc}(c) \mid c \in S\} \mid S \in Uc \setminus \{uc\}\} \cup ptees(N^{uc}, E^{uc})$$

The complete unification procedure is captured by $\text{unifyCells}(Uc, C, E)$, which recursively processes the worklist until no unifications remain.

$$\text{unifyCells}(Uc, C, E) \triangleq \begin{cases} \text{unifyCells}(Uc^{uc}, C^{uc}, E^{uc}) & \text{given any } uc \in Uc \\ (C, E) & \text{otherwise} \end{cases}$$

Termination is guaranteed by observing that each unification reduces either $|C|$ or $|Uc|$. To demonstrate this, note that $|C^{uc}| \leq |C|$ given each unification must alter C such that a single cell represents all cells in uc . Consequently, $|C^{uc}| = |C|$ is only possible given a singleton uc . In this case, $\text{ptees}(uc) = \emptyset$ and Uc must decrease given the removal of uc . As both of these notions are finite, the process must eventually terminate.

It can be shown that unifyCells produces a wellformed data structure graph in which all accessed memory, $\gamma_{SV}(sv, \rho)$ for $\llbracket \text{eval}_{sva}^{\#}(e) \rrbracket_{sz} \Leftrightarrow - \in Cstr, sv \in \text{eval}_{sva}^{\#}(e)$, is represented, establishing **Inv 2** while maintaining **Inv 1**.

Application of the map function as defined above is obviously impractical, as it requires traversing the full graph per unification. To attain an efficient implementation, we employ a *labelled* union-find data structure [12, 19] over cells, in which unions between cells are annotated with an integer to encode positional offsets. For each unification set uc , we create a fresh node n_{new} with cells N^{uc} and union each cell c involved in the unification to its corresponding parent $c' \in N^{uc}$ in the union-find with an label $\delta_{uc}(c.n)$. Accessing a cell traverses the union-find structure, summing the labels to derive c 's full adjustment. This allows the representative of any cell and its applied offset to be determined lazily via the find operation, avoiding the need to update all cells and edges at each step.

5 Evaluation

Table 1 displays our results for a number of target binaries, compiled and decompiles to **Basil** IR code with *Stmts* IR statements, and with the last four being taken from the **TeaDsa** test suite [15]. Since **TeaDsa** targets LLVM, whose types generate different memory structures to our lifted **ARMv8** binaries, we find it infeasible to generate meaningfully comparable metrics between the two. The same holds for existing memory analyses of binaries, since these target x86 architectures and do not generate DSGs.

We evaluate two versions of **DSABin**: One with no extra flags enabled (*-dsa-default*), and one with global splitting and equivalence classes enabled (*-dsa-split -dsa-eqv*). We recorded the following metrics: *Nodes*, the total number of nodes generated across all DSGs; *Collapsed*, the number of these nodes that are collapsed; *Max Cell Density*, the maximum density for a single cell in any DSG, where *density* is the proportion of all memory accesses in the program that target that cell; and *Time*, the running time of our analysis, computed as the average of three runs on a 2.3GHz Quad-Core Intel Core i5 processor.

Increased precision is indicated by a higher number of nodes, fewer collapsed nodes and lower maximum cell density. For **eq** and **stack**, enabling the flags

DSABin Evaluation Results									
Program	Stmts	<i>-dsa-default</i>				<i>-dsa-split -dsa-equiv</i>			
		Nodes	Collapsed	Max Cell Density	Time (ms)	Nodes	Collapsed	Max Cell Density	Time (ms)
eq	19	8	1 (13%)	100%	154	30	0 (0%)	22%	169
split	9	9	2 (22%)	40%	143	28	2 (7%)	40%	152
stack	24	62	1 (2%)	89%	160	98	0 (0%)	22%	192
callgraph1	79	120	12 (10%)	48%	242	205	7 (3%)	48%	256
callgraph2	107	89	1 (1%)	24%	210	157	0 (0%)	24%	247
callgraph3	108	89	1 (1%)	24%	211	157	0 (0%)	24%	258
callgraph4	109	130	2 (2%)	21%	242	220	1 (0.5%)	21%	277

Table 1: Evaluation results for DSABin

resulted in a marked decrease in maximum cell density. In the other examples, the maximum cell density was not affected, but the number of collapsed nodes was reduced, despite significant increases in the total number of nodes.

To assess scalability, we also tested our prototype implementation⁷ of the analysis on `cntlm` [22], an NTLM authentication proxy comprised of threaded C code, which compiles and decompiles to `Basil` IR programs with 46,603 IR statements. On the default mode, the analysis took less than 5 minutes, but retained no global field-sensitivity at all, with all global nodes being collapsed by the end of the top-down phase. Furthermore, 21 out of the 144 procedures that have their own stack lost field-sensitivity, with their respective stack nodes collapsed. With *-dsa-split* enabled, the analysis took 6 minutes and was able to identify 264 distinct global cells in each DSG. However, neither flag had an impact on the number of collapsed stacks.

6 Related work

Closest to our work is the work by Lattner et al. [18] and Gurfinkel et al. [13,15] who apply DSA to LLVM IR but they take different approaches to unification. Lattner’s work [18] requires structural type alignment between objects (nodes) being unified. Gurfinkel et al. [13] improves upon [18] by allowing objects with varying structural types to be unified without the loss of field-sensitivity. However, [13] will still collapse and lose field sensitivity if the aligned cells have different types. Kuderski et al. [15] improves on the TD phase of the DSA.

Pointer analysis on binaries means resolving targets of memory writes. Value set analysis (VSA) is an over-approximating approach to compute potential write targets (i.e., pointers). Balakrishnan et al. [3,4] who coined the term VSA, targets stripped binaries which, in contrast to our binaries, lack the symbol table information used in `Basil`’s DSA. To maintain scalability Balakrishnan’s approach is a fairly coarse over-approximation of the memory layout and leads to many

⁷ This prototype implementation is not optimised in terms of its performance and has room to improve its efficiency.

false positives. Other implementations [6,23,2,14] adapt the abstract domain for particular use cases, e.g., Angr uses a signedness-agnostic domain based on [20], and customise the memory model, e.g., BPA [14] uses a block memory model to support CFG reconstruction through their context-insensitive approach. Our pre-local analysis performs an abstract VSA in which the value set domain can be instantiated to various levels of precision. Future work could investigate, e.g., the benefits of a signedness-agnostic domain for value sets as used in Angr.

Verbeek et al. [25] define and verify an abstract interpretation framework for context-sensitive pointer analysis in which the abstract domains are parametric. Their work investigates three abstract domains and their effects on a pointer analysis for binaries, ranging from precise to very coarse and scalable. A major assumption of their approach is that overlapping memory accesses do not concern pointers. Moreover, not all indirect calls may be resolved in their approach, which results potentially in an under-approximation.

7 Conclusion and future directions

The unification scheme of our approach to data structure analysis, **DSAbin**, is an effective approach for points-to analysis on binaries. We have demonstrated the approach with an implementation in the **Basil** framework and evaluated the results for medium-size binaries by comparing different options for the unification.

The loss of precision due to locally unknown offset accesses, however, is difficult to overcome. To avoid whole stack collapses, splitting of stack regions could be enabled via assumptions on the stack layout. Subsequently, **Basil**'s backend verifier **Boogie** can be tasked to validate these assumptions. When the unknown offset is a parameter or a loop iterator, it is possible to rely on specification, such as preconditions or loop invariants to bound the scope of the resulting collapse. Alternatively, data structure nodes may be extended to represent accesses to variable offsets or with variable sizes. Similar effects may be achieved by propagating constraints to the caller context, effectively inlining local constraints into their callers. Future efforts will investigate these avenues.

Acknowledgments. The authors want to thank Graeme Smith and the anonymous reviewers for their insightful feedback to this work. This research has been supported by the Commonwealth of Australia as represented by the Defence Science and Technology Group of the Department of Defence.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. ARM: Arm architecture reference manual supplement - armv8, for armv8-r AArch64 architecture profile (2022), <https://developer.arm.com/documentation/ddi0600/ac>

2. Balakrishnan, G., Gruian, R., Reps, T.W., Teitelbaum, T.: Codesurfer/x86-a platform for analyzing x86 executables. In: Bodík, R. (ed.) *Compiler Construction*, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. *Lecture Notes in Computer Science*, vol. 3443, pp. 250–254. Springer (2005). https://doi.org/10.1007/978-3-540-31985-6_19, https://doi.org/10.1007/978-3-540-31985-6_19
3. Balakrishnan, G., Reps, T.W.: Analyzing memory accesses in x86 executables. In: Duesterwald, E. (ed.) *Compiler Construction*, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings. *Lecture Notes in Computer Science*, vol. 2985, pp. 5–23. Springer (2004). https://doi.org/10.1007/978-3-540-24723-4_2, https://doi.org/10.1007/978-3-540-24723-4_2
4. Balakrishnan, G., Reps, T.W.: WYSINWYX: what you see is not what you execute. *ACM Trans. Program. Lang. Syst.* **32**(6), 23:1–23:84 (2010). <https://doi.org/10.1145/1749608.1749612>, <https://doi.org/10.1145/1749608.1749612>
5. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) *Formal Methods for Components and Objects*, 4th International Symposium, FMCO 2005. *Lecture Notes in Computer Science*, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17
6. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: Gopalakrishnan, G., Qadeer, S. (eds.) *Computer Aided Verification - 23rd International Conference, CAV 2011. Lecture Notes in Computer Science*, vol. 6806, pp. 463–469. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_37
7. Coughlin, N., Michael, A., Lam, K.: Lift-offline: Instruction lifter generators. In: Giacobazzi, R., Gorla, A. (eds.) *Static Analysis - 31st International Symposium, SAS 2024. Lecture Notes in Computer Science*, vol. 14995, pp. 86–119. Springer (2024). https://doi.org/10.1007/978-3-031-74776-2_4, https://doi.org/10.1007/978-3-031-74776-2_4
8. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL'77)*. pp. 238–252. ACM (1977). <https://doi.org/10.1145/512950.512973>, <https://doi.org/10.1145/512950.512973>
9. Cytron, R.G., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **13**, 451 – 490 (1991), <https://api.semanticscholar.org/CorpusID:13243943>
10. D'Silva, V., Payer, M., Song, D.X.: The correctness-security gap in compiler optimization. In: 2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015. pp. 73–87. IEEE Computer Society (2015). <https://doi.org/10.1109/SPW.2015.33>
11. Feautrier, P.: Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* **20**, 23–53 (1991), <https://api.semanticscholar.org/CorpusID:5738544>

12. Frühwirth, T.W.: Quasi-linear-time algorithms by generalisation of union-find in CHR. In: Fages, F., Rossi, F., Soliman, S. (eds.) Recent Advances in Constraints, 12th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming, CSCP 2007, Rocquencourt, France, June 7-8, 2007, Revised Selected Papers. Lecture Notes in Computer Science, vol. 5129, pp. 91–108. Springer (2007). https://doi.org/10.1007/978-3-540-89812-2_7, https://doi.org/10.1007/978-3-540-89812-2_7
13. Gurfinkel, A., Navas, J.A.: A context-sensitive memory model for verification of C/C++ programs. In: Ranzato, F. (ed.) Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10422, pp. 148–168. Springer (2017). https://doi.org/10.1007/978-3-319-66706-5_8, https://doi.org/10.1007/978-3-319-66706-5_8
14. Kim, S.H., Sun, C., Zeng, D., Tan, G.: Refining indirect call targets at the binary level. In: 28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021. The Internet Society (2021), <https://www.ndss-symposium.org/ndss-paper/refining-indirect-call-targets-at-the-binary-level/>
15. Kuderski, J., Navas, J.A., Gurfinkel, A.: Unification-based pointer analysis without oversharing. In: Barrett, C.W., Yang, J. (eds.) 2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019. pp. 37–45. IEEE (2019). <https://doi.org/10.23919/FMCAD.2019.8894275>, <https://doi.org/10.23919/FMCAD.2019.8894275>
16. Kästner, D., Wünsche, U., Barrho, J., Schlickling, M., Schommer, B., Schmidt, M., Ferdinand, C., Leroy, X., Blazy, S.: CompCerts: Practical experience on integrating and qualifying a formally verified optimizing compiler. In: ERTS 2018: Embedded Real Time Software and Systems. SEE (2018)
17. Lam, K., Coughlin, N.: Lift-off: Trustworthy armv8 semantics from formal specifications. In: Nadel, A., Rozier, K.Y. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2023. pp. 274–283. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_36, https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_36
18. Lattner, C., Lenharth, A., Adve, V.S.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. pp. 278–289. ACM (2007). <https://doi.org/10.1145/1250734.1250766>, <https://doi.org/10.1145/1250734.1250766>
19. Lesbre, D., Lemerre, M., Hara, H.R.A.E., Bobot, F.: Relational abstractions based on labeled union-find. *Proc. ACM Program. Lang.* **9**(PLDI), 1194–1219 (2025). <https://doi.org/10.1145/3729298>, <https://doi.org/10.1145/3729298>
20. Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Signedness-agnostic program analysis: Precise integer bounds for low-level code. In: Jhala, R., Igarashi, A. (eds.) Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7705, pp. 115–130. Springer (2012). https://doi.org/10.1007/978-3-642-35182-2_9, https://doi.org/10.1007/978-3-642-35182-2_9
21. PAC: Home · UQ-PAC/basil wiki (2025), <https://github.com/UQ-PAC/BASIL/tree/main/docs>
22. Sebastian: Cntlm, <https://github.com/versat/cntlm>

23. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Krügel, C., Vigna, G.: SOK: (state of) the art of war: Offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016. pp. 138–157. IEEE Computer Society (2016). <https://doi.org/10.1109/SP.2016.17>, <https://doi.org/10.1109/SP.2016.17>
24. Steensgaard, B.: Points-to analysis in almost linear time. In: Boehm, H., Jr., G.L.S. (eds.) Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 32–41. ACM Press (1996). <https://doi.org/10.1145/237721.237727>, <https://doi.org/10.1145/237721.237727>
25. Verbeek, F., Shokri, A., Engel, D., Ravindran, B.: Formally verified binary-level pointer analysis. In: 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025. pp. 42–53. IEEE (2025)
26. Xu, J., Lu, K., Du, Z., Ding, Z., Li, L., Wu, Q., Payer, M., Mao, B.: Silent bugs matter: A study of Compiler-Introduced security bugs. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 3655–3672. USENIX Association, Anaheim, CA (Aug 2023), <https://www.usenix.org/conference/usenixsecurity23/presentation/xu-jianhao>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Java Analysis



jMT: Testing Correctness of Java Memory Models

Lukas Panneke^(✉)  and Heike Wehrheim 

Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany
{lukas.panneke,heike.wehrheim}@uol.de

Abstract. According to common belief, the Java memory model is broken. In the past, several approaches have proposed repairs, often only to find new programs exhibiting unexpected, unintuitive behavior or the model forbidding standard compiler optimizations. The complexity of defining a memory model for concurrent Java lies in the fact that it requires a *multi-execution model*. Multi-execution models need to inspect many potential executions of a program in order to find the valid ones. Tools automatically validating novel proposals of Java memory models are, however, largely lacking.

To alleviate this problem, we introduce jMT, a novel tool for constructing multi-execution semantics for concurrent Java programs. jMT relies on single-execution models defining *well-formed* execution graphs, based on which it builds valid multi-execution semantics via causality checking. Thereby, jMT supports evaluating new proposals of Java memory models (JMMs) on a per-program basis. jMT can furthermore be employed for testing the conformance of JMMs to existing compilation schemes and compilers. Our evaluation of jMT on 169 litmus tests reveals a number of interesting insights into existing JMMs.

Keywords: Java memory model, multi-execution models, compilation

1 Introduction

In the past, there have been numerous efforts for correctly defining the Java Memory Model (JMM) [4, 17, 23], i.e., defining the semantics of concurrent Java programs on multiprocessor machines. The official 1996 specification [8] has been shown to be flawed by Pugh [29]. The next 2004 proposal of a JMM (Manson et al. [23], in the sequel called JLS₀₄) has also been criticized for being incompatible with some desired compiler optimizations [5, 31]. Moreover, it is outdated as far as modern synchronization primitives are concerned, in particular the features of the `VarHandle` API. The `VarHandle` API adds new access modes for variables (opaque and release/acquire), read-modify-write operations, memory fences and the mixing of volatile and non-volatile accesses to the same location. The intended semantics of this API has been extensively described by its designer Doug Lea [16]. The most recent JMM proposal of Bender, Liu and Palsberg [4, 17] (in the following JAM₂₁) incorporates most of these new features, the

proposed model – however – also forbids certain expected, observable behavior, and allows undesirable behavior, as we show below.

The core reason for this difficulty in defining a correct JMM lies in the fact that Java requires a *multi-execution model*: the validity of executions (i.e., whether some particular execution and its outcome should be allowed or not) cannot be determined by looking at one execution in isolation. Instead, given a set of candidate executions, a subset of valid executions is to be constructed. Java memory models therefore often consist of two parts: a single-execution model (SEM) defining a set of candidates and on top of that a multi-execution model (MEM) used for validating the executions in the candidate set. As Batty et al. [3] have discussed within the context of C++, single-execution models alone can never suffice without restricting compiler optimizations. Basically, the reason for this is the so called “thin-air-problem” in which values (of shared locations) circularly depend on their own computation.

Given the past experiences in constructing new JMMs, every new proposal first has to undergo several healthiness checks: (1) evaluating the novel semantics on numerous *litmus tests* (small programs with interesting behavior), (2) checking agreement with standard compilation schemes and hardware memory models, and (3) validating the conformance of compilers with the semantics. Currently, in particular check (2) is ultimately carried out via formal proving [15, 18, 27]. Proofs are, however, too labour-intensive to be performed on every change of the model. A tool allowing to first of all *test* such properties would enable fast iteration of model changes so that expensive proving can be deferred to later stages of the model development, once testing finds no further inconsistencies.

Currently, the most prominent tool supporting the development of new memory models is HERD: HERD allows for the definition of single-execution models, independent of any specific programming language or hardware. HERD does, however, not support multi-execution models. A second potentially usable tool is JAVASIM [21] which is however known to contain bugs.

In this paper, we therefore introduce the novel tool *jMT* for multi-execution models. *jMT* allows for *testing* the above-mentioned healthiness checks for novel Java memory models. To this end, a user of the tool has to develop a new single-execution model for Java and specify it in the *.cat* language of HERD. *jMT* then automatically calculates the semantics of concurrent programs, thereby employing *jMT*’s multi-execution model which builds on JLS₀₄’s MEM (as this still remains the official JMM). We provide an operational formulation of the JLS₀₄-style causality semantics, turning its declarative definition into an executable construction of justification sequences. Contrary to JLS₀₄, our multi-execution model is based on the concept of *symbolic execution graphs*. Such graphs allow us to efficiently represent infinitely many candidate executions within a single graph, and to solve parts of the justification search using an SMT solver. Symbolic execution graphs have also in parallel been developed in [30], however, are therein employed for different forms of justifications.

jMT can be used in three different scenarios: **1** to generate the semantics of litmus tests and compare the outcomes against expected outcomes, **2** to

compile programs using standard compilations schemes to hardware and contrast the outcomes of litmus tests on the new JMM with outcomes of the compiled program, and **3** to compile programs using standard Java compilers and then once more compare outcomes. To support these three application scenarios, we provide – along with jMT – a collection of 169 litmus tests, assembled from [4, 9, 13–16, 18–20, 22, 28, 31–33]. We have furthermore implemented a tiny compiler to Intel’s x86 processor to support scenario **2**. This compiler does not perform any optimizations, so that a new JMM can be checked with respect to a compilation scheme alone. Finally, to support scenario **3** our tool includes a toolchain in which programs can be compiled using standard Java compilers. For this purpose we employ a Java concurrency stress testing tool (JCSTRESS [33]) to enable finding mismatches in behaviors at runtime.

We have implemented and experimentally evaluated jMT. Our evaluation shows that jMT is able to find surprising novel issues with existing JMMs and that jMT is – according to our testing and given the right SEM – an accurate simulator for JLS₀₄.

Related Work. Several tools have been proposed for simulation, model checking, and compilation validation under weak memory. JAVASIM [21] is an early simulator for the Java memory model. It was developed alongside JLS₀₄ and predates the `VarHandle` APIs. Consequently, it lacks support for many current language features. Discrepancies between its results and JLS₀₄ further limit its applicability. Many other multi-execution models have tool support as well [10, 24, 26, 30]. Those tools support our scenario **1** for their respective memory model, instead of the Java memory model. Many more tools target single-execution models.

HERD [1] and DARTAGNAN [6] are prominent tools for evaluating litmus tests for `.cat`-specified models, but since all `.cat` models are *single-execution*, they cannot directly support multi-execution semantics as in JLS₀₄. JMC [11] and GENMC [12] are memory model parametric model checkers requiring acyclic $\text{po} \cup \text{rf}$, excluding JLS₀₄, which does allow such cycles in the presence of data races. They can still analyze Java programs proven race-free by other means.

We are, to our knowledge, the first to apply testing scenarios **2** and **3** to *multi-execution* models. TÉLÉCHAT [7] detects weak memory bugs in optimizing production compilers using static analysis with HERD, whereas we use a non-optimizing compiler in scenario **2** and dynamic analysis in scenario **3**. TRICHECK [34] – similarly to jMT – validates compilation schemes, but for the *single-execution* C++ model.

Limitations. Our tool supports all features of the `VarHandle` API, but lacks support for other aspects of the Java programming language relevant to visibility and ordering: monitors, external events (such as I/O), final fields, and observations of thread lifetime (first and last event, start and termination detection). We furthermore do not support `.cat` models with mutually recursive relations.

Contributions. Beyond tool implementation, this paper makes the following contributions: (1) We introduce symbolic execution graphs. (2) We are (to our

<i>values</i>	$v \in \text{Val} = \{0, 1, \dots\}$	<i>locations</i>	$x, y \in \text{Loc} = \{x, y, \dots\}$
<i>local registers</i>	$a \in \text{Reg} = \{a, b, \dots\}$	<i>thread identifiers</i>	$\tau \in \text{TID} = \{1, 2, \dots\}$
<i>read modes</i>	$\text{rm} \in \{\text{pln}, \text{opq}, \text{acq}, \text{vol}\}$	<i>write modes</i>	$\text{wm} \in \{\text{pln}, \text{opq}, \text{rel}, \text{vol}\}$
<i>fence modes</i>	$\text{fm} \in \{\text{WW}, \text{RR}, \text{acq}, \text{rel}, \text{full}\}$		

$$\begin{aligned}
\text{exp} &::= a \mid v \mid \text{exp} + \text{exp} \mid \text{exp} = \text{exp} \mid \text{exp} > \text{exp} \mid \neg \text{exp} \mid \text{exp} \wedge \text{exp} \mid \dots \\
\text{sxp} &::= a \mid v \\
c &::= a := \text{exp} \mid \text{STORE}^{\text{wm}}(x, \text{sxp}) \mid a := \text{LOAD}^{\text{rm}}(x) \mid \text{fence}^{\text{fm}} \mid a := \text{CAX}^{\text{rm}, \text{wm}}(x, \text{sxp}, \text{sxp}) \\
C &::= \text{SKIP} \mid c \mid C ; C \mid \text{if } \text{exp} \text{ then } C \text{ else } C
\end{aligned}$$

Fig. 1: Syntax of sequential programs.

knowledge) the first to formalize the cross-execution event equivalence, a key but underspecified part of JLS₀₄'s MEM. (3) We show that JAM₂₁ is flawed via counterexamples, including mismatches between intended and actual behavior, and a counterexample to its claimed compilation correctness to x86. (4) We identify a contradiction in Lea's VarHandle documentation [16], the de facto source for volatile semantics in presence of access mode mixing on the same variable.

2 Foundations

We start by introducing some basic definitions and notations.

Syntax of programs. Figure 1 defines the syntax of sequential programs¹. We do not use Java syntax here, just plain pseudocode syntax. Sequential programs consist of assignments to registers, store (write) and load (read) instructions, writing to and reading from shared locations as well as a read-modify-write (RMW²) instruction Compare-And-eXchange. We assume programs to be in static single assignment form and registers to be disjoint between threads. Programs might furthermore contain fences. Stores, loads, RMWs and fences come with *access modes* (as of the VarHandle API). A concurrent program P is then a parallel composition (denoted $\parallel$) of sequential programs, one for each thread in TID. We do not give a semantics here; the semantics is fixed by a single execution model (SEM).

Execution of programs give rise to execution graphs in which the nodes are *events*³ and edges describe relations over events. Events are labelled over a set of labels **Lab**. In our multi-execution model, we build *symbolic* execution graphs in which the labels contain symbolic expressions *sxp* (register names or constants) instead of concrete values only, i.e.,

$$\begin{aligned}
\text{Lab} = \{ & \text{R}^{\text{rm}}(x, \text{sxp}_R), \text{W}^{\text{wm}}(x, \text{sxp}_W), \text{RMW}^{\text{rm}, \text{wm}}(x, \text{sxp}_R, \text{sxp}_W), \text{F}^{\text{fm}} \\
& \mid \text{sxp}_R, \text{sxp}_W \in \text{Val} \cup \text{Reg} \} .
\end{aligned}$$

¹ Alike other tools for weak memory models, our tool does not consider loops.

² As RMW, we include a compare-and-exchange instruction; others are also possible.

³ In JLS₀₄, these are called actions.

The functions `typ`, `loc`, `sxpR` and `sxpW` return (when applicable) the type (R/W/RMW/F), location, read and written symbolic expression of a given label. An execution of an instruction by a thread $\tau \in \text{TID}$ leads to an *event* labelled via `Lab` (plus with a unique identifier). Initializing writes are specific events: They have no access modes and are performed by a special thread `ini` (initializing events of the form $W_{\text{ini}}(x, 0)$)⁴. Together, the set of events is $\text{Events} = (\text{Lab} \times \text{TID} \times \mathbb{N}) \cup \{W_{\text{ini}}(x, 0) \mid x \in \text{Loc}\}$. To access the thread of an event e , we write $\text{tid}(e)$. We let $R \triangleq \{e \mid \text{typ}(e) \in \{R, \text{RMW}\}\}$, and $W \triangleq \{e \mid \text{typ}(e) \in \{W, \text{RMW}\}\}$, be the set of read and write events, respectively.

Notation. Within execution graphs, events get ordered via various relations. For relations R, R_1, R_2 we write R^{-1} for the inverse relation of R and $R_1; R_2$ for the relational composition of R_1 and R_2 . We write R^+, R^* and R^- for the transitive and the reflexive-transitive closure and the transitive reduction of R , respectively. For a set S , let $S|_\tau \triangleq \{e \in S \mid \text{tid}(e) = \tau\}$ restrict S to thread τ , and $R|_\tau \triangleq R \cap (\text{Events}|_\tau \times \text{Events}|_\tau)$ restrict R to pairs of events with thread τ . Finally, $R|_S \triangleq R \cap (S \times S)$ restricts R to pairs over the events of S .

Litmus tests. From a set of execution graphs, we can see the *behavior* of a program, i.e., the possible values of registers at the end of the program. Formally, behaviors are mappings $\beta : \text{Reg} \rightarrow \text{Val}$ and are specified via assertions (logical formulae) over registers and values. For such a logical formula φ , we write $\mathcal{B}_\varphi := \{\beta \mid \beta \models \varphi\}$ for the set of behaviors described by φ . A litmus test in the tool HERD consists of two parts: a program and a behavior assertion. We can have two sorts of behavior assertions: *required* and *forbidden* behaviors.⁵ A set of behaviors $\mathcal{B}$ is *consistent* with a set of required and forbidden behaviors φ ($\mathcal{B} \models \varphi$) if all of the required and none of the forbidden behaviors are in $\mathcal{B}$.

3 Symbolic Execution Graphs

Executions of programs are modelled via (symbolic or concrete) *execution graphs* with events as nodes and edges describing relations. Common to all JMMs are the relations program order `po` (order of steps within one thread as per the program) and reads-from `rf` (to see which writes have written the values read by reads). Typically, memory models require `po` and `rf` together to be acyclic, as to forbid out-of-thin-air-behavior. The Java memory model, however, allows $\text{po} \cup \text{rf}$ to contain cycles. As an illustration of this fact (and the way we describe such executions within symbolic execution graphs), consider the following program `LbOdd` (taken from [20, Fig. 1.3]).

$$\begin{array}{l|l} \mathbf{x} = \mathbf{y} = 0 & \\ \mathbf{a} := \text{LOAD}^{\text{pln}}(\mathbf{x}); & \mathbf{c} := \text{LOAD}^{\text{pln}}(\mathbf{y}); \\ \mathbf{b} := \mathbf{a} \mid 1; & \text{STORE}^{\text{pln}}(\mathbf{x}, \mathbf{c}) \\ \text{STORE}^{\text{pln}}(\mathbf{y}, \mathbf{b}) & \end{array}$$

⁴ For simplicity, we assume initial values to be 0.

⁵ For the reader familiar with HERD, required means we require that HERD reports that the final state assertion holds **sometimes** or **always**, forbidden that it holds **never**.

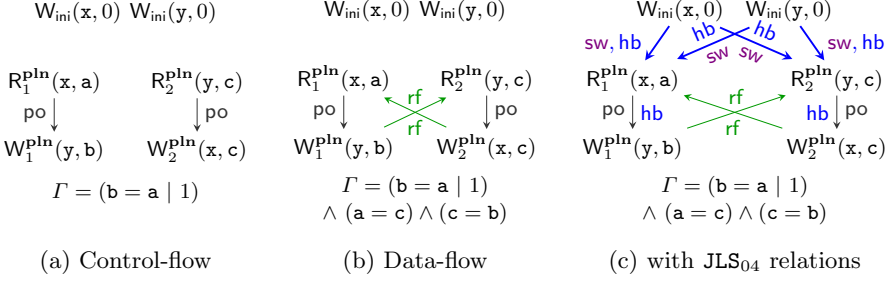


Fig. 2: An example of stages 1–3 of the algorithm, constructing the symbolic execution graph for program `LbOdd`. Single thread control flow semantics a, one symbolic execution graph (b), and its enhancement (c) with relations of JLS_{04} .

Here, all access modes are plain (**pln**), and $|$ is the bitwise logical “or” operator. The behavior $a = 1 \wedge b = 1 \wedge c = 1$ is allowed by the Java language specification [9] for the following reason: Interthread analysis could determine that x and y are always either 0 or 1, and thus determine that b is always 1. If the constant 1 is substituted for b , then this example becomes a classic load buffering example, where the behavior must be allowed to be efficiently implementable on e.g. ARM processors. Figure 2b describes the symbolic execution graph out of which we can derive such a behavior. In this graph, we see symbolic expressions (register names) instead of concrete values being read and written. Attached to this graph, we have a *constraint*, a logical expression over symbolic expressions detailing dependencies. For the graph in Figure 2b, the constraint would be

$$\Gamma = \underbrace{b = a \mid 1}_{\text{assign. constraint}} \quad \wedge \quad \underbrace{a = c}_{\text{rf-constraint}} \quad \wedge \quad \underbrace{c = b}_{\text{rf-constraint}}$$

Note that we can use register names in symbolic expressions because of the static single assignment form of programs.

Definition 1. A symbolic execution graph $G = (E, \text{po}, \text{rf}, \Gamma)$ of a program P has the following components:

- $E \subseteq \text{Events}$ is a set of events containing exactly one initial write $W_{\text{ini}}(x, 0)$ for every location x used in P .
- $\text{po} \subseteq E \times E$ is the program order totally ordering the events within a thread, i.e., $\forall \tau \in \text{TID} : \text{po}|_{\tau}$ is a total order on $E|_{\tau}$.
- $\text{rf} \subseteq R \times W$ is the reads-from relation satisfying $\text{loc}(e) = \text{loc}(e')$ for all $(e, e') \in \text{rf}$. Each read reads-from exactly one write: rf^{-1} is a function.
- Γ is a set (or conjunction) of constraints over symbolic expressions such that $(\text{sxp}_R(e') = \text{sxp}_W(e)) \in \Gamma$ for all $(e, e') \in \text{rf}$.

For a graph G , we write $G.E$ for its events, $G.W$ for its write events, $G.\text{po}$ for its program order and so on. A concrete execution graph is a symbolic execution

graph where each expression is a constant and each read is justified by a write with the same value, that is, $\forall(e, e') \in \mathbf{rf} : \mathbf{sxp}_W(e) = \mathbf{sxp}_R(e') \in \mathbf{Val}$. We obtain a concrete execution graph from a symbolic execution graph by finding a mapping $\gamma : \mathbf{Reg} \rightarrow \mathbf{Val}$ which is a logical model of Γ and then substituting all symbolic expressions in events by their instantiations via γ .

Building execution graphs. Similar to HERD, we build execution graphs in multiple stages. However, while HERD attempts to build execution graphs with concrete values, and fails to properly evaluate assertions for graphs where no unique value can be found by forward substitution, we use symbolic execution graphs to handle these cases, as they are highly relevant to MEMs, specifically JMMs.

1. Control-flow semantics. First, we construct graphs per thread by following the thread's program according to a sequential semantics. Initially, the thread graph is empty, i.e., $G_0 = (\emptyset, \emptyset, \emptyset, \emptyset)$. Given a graph $G = (E, \mathbf{po}, \mathbf{rf}, \Gamma)$ representing the execution so far, we extend it depending on the type of the next instruction of thread τ . All newly added events are appended $\mathbf{po}$ -after the previous events. Relation $\mathbf{rf}$ remains empty in this stage.

Assignment For an assignment of the form $a := \mathit{exp}$, we extend Γ by the constraint $(a = \mathit{exp})$. No additional events are added.

Store For a store $\mathbf{STORE}^{\mathbf{wm}}(x, \mathit{sxp})$, we add an event labeled $W^{\mathbf{wm}}(x, \mathit{sxp})$ with thread id τ and a fresh identifier, without adding constraints.

Load For a load $a := \mathbf{LOAD}^{\mathbf{rm}}(x)$, we add an event labeled $R^{\mathbf{rm}}(x, a)$, with thread id τ and – as for all events – a fresh identifier, without adding constraints.

RMW For a compare-and-exchange instruction $a := \mathbf{CAX}^{\mathbf{rm}, \mathbf{wm}}(x, \mathit{sxp}_1, \mathit{sxp}_2)$, we have to construct two possible extensions, one where the CAX succeeds and the other where it fails. In the first case, we add an event labelled $\mathbf{RMW}^{\mathbf{rm}, \mathbf{wm}}(x, \mathit{sxp}_1, \mathit{sxp}_2)$ and a constraint of the form $(\mathit{sxp}_1 = \mathit{sxp}_2)$, in the second case an event labelled $R^{\mathbf{rm}}(x, \mathit{sxp}_1)$ and a constraint $(\mathit{sxp}_1 \neq \mathit{sxp}_2)$.

Fence For a fence $\mathbf{fence}^{\mathbf{fm}}$, we add an event labelled $\mathbf{F}^{\mathbf{fm}}$ and no constraints.

If-then-else Like for RMW, we build two successor graphs, one with additional constraint cond (being the condition of the if) and one with $\neg \mathit{cond}$.

The procedure ends once no instructions remain. Since the focus of weak memory models is on values read from and written to shared locations, termination of the procedure requires the absence of loops whose behavior depends on memory accesses. Consequently, our tool, like most other weak-memory tools, supports only loop-free programs. At that point, we have all conceivable executions of a thread, i.e., for every thread τ we have a set of graphs $\mathcal{G}_\tau$. The program executions of the parallel composition of the threads then correspond to the Cartesian product of the graphs of all threads. More precisely, for each family of graphs $\langle G_\tau = (E_\tau, \mathbf{po}_\tau, \emptyset, \Gamma_\tau) \rangle_{\tau \in \mathbf{TID}}$, one per thread, we construct a new product graph $G = (\bigcup_{\tau \in \mathbf{TID}} E_\tau, \bigcup_{\tau \in \mathbf{TID}} \mathbf{po}_\tau, \emptyset, \bigcup_{\tau \in \mathbf{TID}} \Gamma_\tau)$. To this graph, we finally add initializing events $W_{\mathbf{ini}}(x, 0)$ for every $x \in \mathbf{Loc}$.

An example of the result of stage 1 is shown in Figure 2a. Since the LbOdd program has no branching and no compare-and-exchange instructions, this is the only result.

2. Data-flow semantics. In stage 2, we add the **rf** edges. Out of every graph $G = (E, \text{po}, \text{rf}, \Gamma)$ which stage 1 produces, we construct several graphs by adding all possible **rf** relations. More specifically, for every $r \in E \cap R$, we choose some $w \in E \cap W$ such that $\text{loc}(r) = \text{loc}(w)$, add (w, r) to **rf** plus add a constraint $(\text{exp}_R(r) = \text{exp}_W(w))$ to Γ . Whenever the resulting set of constraints Γ is unsatisfiable (which is checked using an SMT solver), we entirely remove the graph from our set of considered graphs. Note that we have no further requirements on **po** and **rf** here; this is what a concrete single-execution model adds.

One of the graphs created by stage 2, where each read reads from a non-initial write, is shown in Figure 2b. As the result of stage 2, we also get 3 other graphs, one where all read events read from initial writes, and two where one of the read events read from an initial and the other from a non-initial write.

3. Single execution memory model. So far, the constructed graphs are independent of any SEM. In the last step, we now take the set of graphs $\mathcal{G}$ computed as the result of stage 2, a SEM in the form of a **.cat** model of HERD and remove all graphs inconsistent with the requirements of the SEM.

Typically, memory models define more relations than just **po** and **rf**. With a SEM fixed, we thus build *enhanced* symbolic execution graphs. As an example, we use JLS_{04} (the complete **.cat** model can be found in the extended version of this paper). JLS_{04} defines a relation **hb** (happens-before) by $\text{let } \text{hb} = (\text{po} \mid \text{sw})^*$, i.e., defines $\text{hb} = (\text{po} \cup \text{sw})^*$, where **sw** is the relation synchronizes-with. It then states the consistency requirement **irreflexive rf;hb**. A number of other relations and constraints fix further requirements.

Figure 2c shows the enhancement of the graph of Figure 2b with relations of JLS_{04} . Note that – due to the symbolic expressions – this graph and its constraint represent an infinite number of concrete graphs, namely all those where $a = b = c = v$ for some odd value $v \in \text{Val}$.

4 Checking Causality Requirements

Next, we build the multi-execution model out of the set of graphs $\mathcal{G}$ constructed according to a SEM. The overall objective of the construction is to determine which graphs are *valid*, thereby ruling out unwanted out-of-thin-air behavior. As an example, consider the graph in Figure 2c of program **LbOdd**. This graph has a potentially problematic *race* (actually two): Event $W_2^{\text{pln}}(x, c)$ has an **rf** edge to event $R_1^{\text{pln}}(x, a)$ and these two events are not ordered via relation happens-before **hb**. The question to be answered is whether a JMM considers this to be ok or not. To this end, the Java specification has introduced the concept of *causality checking*, which we here extend to symbolic execution graphs.

Our tool expects the SEM to have constructed *well-formed* graphs G for a program containing program order **po** and reads-from relation **rf** plus relations **so** (synchronization order), **sw** (synchronizes-with) and **hb** (happens-before), i.e., $G = (E, \text{po}, \text{so}, \text{rf}, \text{sw}, \text{hb}, \Gamma)$, and the additional relations have to satisfy

- **hb** is a partial order, $\text{sw} \subseteq \text{hb}$ and **so** totally orders the volatile events.

The multi-execution part of the memory model now considers multiple execution graphs to determine the validity of a single graph. To this end, it has to relate events of *different* execution graphs. For this, we define an equivalence relation on events of (potentially different) graphs⁶. First, we consider two events $e_1 \in E_1, e_2 \in E_2$ to be *similar*, denoted $e_1 \sim e_2$, if $\text{typ}(e_1) = \text{typ}(e_2)$ and – if they operate on a location – $\text{loc}(e_1) = \text{loc}(e_2)$. For each event, we count the number of similar events preceding it in program order within G , giving its *index*: $\text{idx}(e, G)$. If $e \in G.E$ is an initial write action, then we define $\text{idx}(e, G) = 0$. Otherwise, e has an executing thread $\tau \in \text{TID}$ and $G.\text{po}|_\tau$ is a total order. We define $\text{idx}(e, G)$ to be the position of e in $G.\text{po}|_\tau \cap \sim$. Finally, events e_1 and e_2 of graphs G_1 and G_2 are *equal* if they are similar and have the same position: $e_1 = e_2 \Leftrightarrow e_1 \sim e_2 \wedge \text{idx}(e_1, G_1) = \text{idx}(e_2, G_2)$. In the sequel all operations on elements, sets and relations are performed *modulo this equivalence relation*.

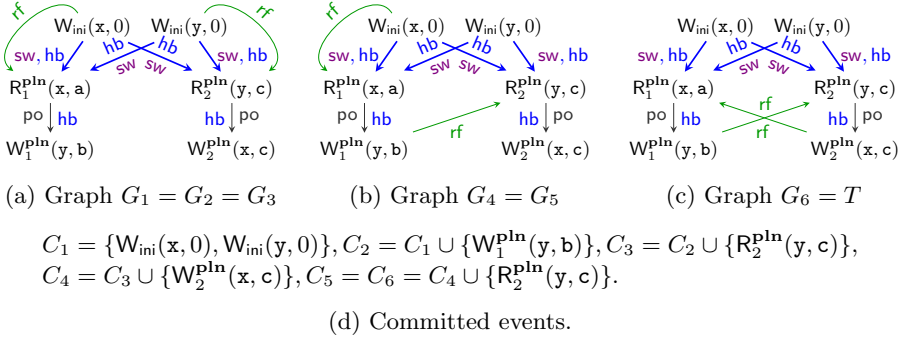
The MEM now takes the set of well-formed execution graphs $\mathcal{G}$ and rules out those that are considered unreasonable. Let $T \in \mathcal{G}$ be a *target* which we intend to check for validity. Causality checking incrementally constructs a *justification* for T from our set of graphs $\mathcal{G}$: we start the construction with a well-behaved execution $G \in \mathcal{G}$ in which all **rf**-related events are also in happens-before relation: $\forall (w, r) \in G.\text{rf} : (w, r) \in G.\text{hb}$. Then, we incrementally *commit* events. Committing basically fixes values written or read by events (in our case, we fix symbolic values). The Java language specification defines a number of conditions on commitments which we have implemented in jMT⁷. The check for validity of T ends with success when we have managed to commit all of T 's events. More formally, a target T is allowed by the memory model, if there is a so called *justification sequence* $G_1, \dots, G_n$ where each G_i is well-formed, together with a sequence of committed events $C_0 \subseteq \dots \subseteq C_n \subseteq \text{Events}$ where $C_0 = \emptyset$, and $C_n = G_n.E$. In addition, the following six conditions (of JLS₀₄) need to hold for every i , which we capture in a predicate $\text{ok}(T, \langle G_i, C_i \rangle, C_{i-1})$:

1. $C_i \subseteq G_i.E$ (committed events occur in the graph),
2. $G_i.\text{hb}|_{C_i} = T.\text{hb}|_{C_i}$ (**hb** over committed events C_i is the same in G_i and T),
3. $G_i.\text{so}|_{C_i} = T.\text{so}|_{C_i}$ (**so** over committed events C_i is the same in G_i and T),
5. $G_i.\text{rf}|_{C_{i-1}} = T.\text{rf}|_{C_{i-1}}$ (**rf** over committed events C_{i-1} agrees in G_i and T),
6. $\forall r \in G_i.R \setminus C_{i-1} : (G_i.\text{rf}^{-1}(r), r) \in G_i.\text{hb}$ (reads not committed in stage $i-1$ must be in **hb**-order with their writes),
7. $\forall r \in (C_i \setminus C_{i-1}) \cap R : G_i.\text{rf}^{-1}(r) \in C_{i-1} \wedge T.\text{rf}^{-1}(r) \in C_{i-1}$ (reads can only be committed after their writes).

The JLS₀₄ additionally has conditions numbered 4, 8 and 9. Condition 9 is about external operations (like print statements) which, for simplicity, we ignore here; Condition 8 is given below and only checked after justifications are generated. Condition 4 requires that $V_i|_{C_i} = V|_{C_i}$ for all i , where V is a function mapping each write w to the written value. Basically, condition 4 is saying that

⁶ To our knowledge, no existing JMM literature formally defines such an equivalence.

⁷ If a user of jMT wishes to use a different set of conditions, the code must be edited. For conditions such as those of [2] this is straightforward.

Fig. 3: Justification sequence for the execution graph T of Figure 2c.

once we have committed an event, its value may not change anymore. Since we have symbolic execution graphs, we might not only have values, but also symbolic expressions. Our tool thus first ignores condition 4 during construction of justification sequences and then checks it in a final validation step (see below).

4. Generating Justifications. Stage 4 in the construction of valid execution graphs out of programs is now the actual generation of justification sequences. Formally, a justification sequence is a sequence of *justification stages* which are pairs $s = \langle G, C \rangle$ where G is an execution graph and $C \subseteq \text{Events}$ is a set of committed events. A *justification* for a target execution graph T is a pair $J = \langle T, (s_1, \dots, s_n) \rangle$ where $(s_1, \dots, s_n)$ is a justification sequence.

For a given target graph T , a current justification stage $\langle G, C \rangle$ and a collection of well-formed execution graphs $\mathcal{G}$, we compute the successor justification stages with $\text{GenSuccessors}(T, \langle G, C \rangle, \mathcal{G}) =$

$$\bigcup_{G' \in \mathcal{G} \text{ } \emptyset \neq X \subseteq G'.E \setminus C} \bigcup_{\{ \langle G', C' \rangle \mid C' = (C \cap G'.E) \cup X \wedge \text{ok}(T, \langle G', C' \rangle, C) \}}.$$

We always start a justification sequence with a *well-behaved* execution $G_1 \in \mathcal{G}$. We set C_0 (which never gets constructed) to $\emptyset$. For graph G_1 , the initial stage is $s_1 = \langle G_1, G_1.E|_{ini} \rangle$, the stage with just the initial writes committed. We then generate the justification sequences by exploring all finite sequences $(s_1, \dots, s_n)$ such that for all $1 < i < n$, $s_{i+1} \in \text{GenSuccessors}(T, s_i, \mathcal{G})$. A justification sequence $(s_1, \dots, \langle G_k, C_k \rangle)$ generated in this way is *valid* (and the target T is valid) iff $|C_k| = |T.E|$ and $\text{ok}(T, \langle T, T.E \rangle, C_k)$ and condition 8 is fulfilled:

8. if $(x, y) \in G.\text{sw} \cap (G_i.\text{hb}^- \setminus G_i.\text{po}) \wedge (y, z) \in G_i.\text{hb} \wedge z \in C_i$, then $(x, y) \in G_j.\text{sw}$ for all $j \geq i$.

Condition 8 enforces the following constraint: once an action z is committed, removing the release-acquire synchronization that was necessary to justify why z is allowed to see what it sees is not possible anymore.

As an example of stage 4, consider program `LbOdd` and one of its symbolic execution graphs T as depicted in Figure 2c. Figure 3 shows the graphs of a

justification sequence for T with corresponding committed events sets. As seen, events are incrementally committed until finally $C_6 = T.E$ and $G_6 = T$.

5. Validating Symbolic Justifications. Next, we need to take condition 4 into account which we carry out together with checking for specific behaviors. Our check has to guarantee that (1) there is some instantiation of the registers occurring in symbolic expressions with values such that values of committed events do not change anymore over a justification sequence, (2) the target T allows for the behavior assertion, and (3) the constraints of $T, T.\Gamma$, are satisfied. The check proceeds by encoding these three properties into a large first-order logic formula and asking an SMT solver for a model of the formula. We encode each read and write of the graphs into a variable of the form *graph_name:thread:type location@index*, for example, $G_2:T_3:Ry@2$ is the variable representing the 2nd read of location y performed by thread T_3 in execution graph G_2 . We encode the entire justification sequence and target T in such a way that the SMT solver finds a satisfying assignment to these variables, whenever an instantiation of symbolic register names in the justification sequence with the values of the corresponding variables yields a concrete justification demonstrating that the behavior as specified in the assertion is possible.

The encoding works as follows. For a register a occurring as symbolic value in a graph G , we define a local variable $ELV_G(a) \triangleq G.a$. We denote by $\phi[ELV_G]$ the replacement of each register in formula ϕ by ELV_G . We encode a non-initializing⁸ write $w \in G_i.W \setminus G_i.E|_{\text{ini}}$ to location x in justification stage $s_i = \langle G_i, C_i \rangle$ by thread τ to location x with write expression $wexpr$ and $\text{idx}(w, G_i) = p$ as

$$\text{EncWr}(w, s_i) \triangleq \begin{cases} G_i:\tau:Wx@p = wexpr[ELV_{G_i}] = T:\tau:Wx@p & \text{if } w \in C_i \\ G_i:\tau:Wx@p = wexpr[ELV_{G_i}] & \text{otherwise.} \end{cases}$$

We encode a read $r \in G_i.R$ in justification stage $s_i = \langle G_i, C_i \rangle$ by thread τ from location x with read expression $rexpr$ and $\text{idx}(r) = p$ as $\text{EncRd}(r, s_i) \triangleq G_i:\tau:Rx@p = rexpr[ELV_{G_i}]$.

We take the conjunction of the encoding of all reads and non-initializing writes in a justification stage, and the graphs constraints Γ , to encode the stage $\text{EncStage}(\langle G_i, C_i \rangle) \triangleq$

$$\bigwedge_{w \in G_i.W \setminus G_i.E|_{\text{ini}}} \text{EncWr}(e, \langle G_i, C_i \rangle) \wedge \bigwedge_{r \in G_i.R} \text{EncRd}(e, \langle G_i, C_i \rangle) \wedge G_i.\Gamma[ELV_{G_i}].$$

We take the conjunction of this encoding for all justification stages of the justification and the target stage. We also take the conjunction with the behavior assertion ϕ substituted with ELV_T , to ensure that the justification works for a concrete target execution satisfying the assertion. This encodes the entire justification $\text{EncJust}(\langle T, (s_1, \dots, s_n) \rangle, \phi) \triangleq$

$$\phi[ELV_T] \wedge \text{EncStage}(\langle T, T.E \rangle) \wedge \bigwedge_{1 \leq i \leq n} \text{EncStage}(s_i).$$

⁸ Initializing writes write a constant that is the same for all executions of a program, so condition 4 cannot be violated.

This formula might have a satisfying assignment mapping each read and write expression to a value. If so, substituting these values into the justification gives a concrete justification satisfying all conditions, including condition 4, showing that a concrete execution satisfying the assertion is acceptable.

For our example program `LbOdd`, we can thus see that the behavior $a = 1 \wedge b = 1 \wedge c = 1$ is possible. Note that an assignment with $b \mapsto 5$ is for instance not possible: in the justification sequence of Figure 3, graph G_2 has the (non shown) constraint $b = 1$. Since $\mathbf{W}_1^{\text{Pln}}(\mathbf{y}, \mathbf{b}) \in C_2$ (i.e., committed), this value has to be kept over the entire justification sequence.

With the construction of symbolic execution graphs and causality checking, we now have all the tooling at hand for application scenario [1](#). Next, we investigate application scenario [2](#).

5 Compilation Scheme Correctness

The second key objective of our work is to validate that a candidate Java memory model is compatible with a realistic compilation scheme and associated hardware memory model. Intuitively, soundness requires no compiled program to exhibit behavior forbidden by its source. Formally, a compilation scheme $Comp$ from a source Java memory model SEM (and its multi-execution version $MEM(SEM)$) to a target hardware model H is sound if, for every source program P ,

$$\text{Beh}_H(Comp(P)) \subseteq \text{Beh}_{MEM(SEM)}(P), \quad (1)$$

where $\text{Beh}_M(P)$ denotes the set of behaviors of P under model M . Our notion of compilation correctness coincides with the intent of [18]: “A correct compilation, intuitively, should not introduce any new program behavior,” although their formalization is different from ours. *Proving* this inclusion is expensive and error-prone (as seen in Section 7). To support fast, iterative development of models, jMT offers an automated *testing-based* approximation. It can expose unsoundness by finding a counterexample, i.e., a behavior of $Comp(P)$ not permitted by $MEM(SEM)$, but cannot prove soundness.

jMT takes as input a source SEM in HERD’s `.cat` format and a program P written in HERD’s litmus test format for Java.⁹ The tool outputs either a *pass* (no counterexamples) or a *fail* verdict, with a witness behavior violating the inclusion. We currently target the x86 architecture due to its simplicity, appearance in [18], and strong HERD support. Instead of using an existing compiler, we implemented a simple, non-optimizing compiler to isolate the compilation scheme itself. We briefly describe this compiler. Each source instruction is translated into a sequence of x86 instructions according to the scheme given in the extended paper,¹⁰ and source registers are systematically mapped to x86 registers. We track which thread each register belongs to, since in x86 each thread has its own register set. For instance, `a` in thread T_1 may map to `EAX` of T_1 , while

⁹ Although the behavior assertion is not of interest in scenario [2](#), we keep this format because that way we can use the same input file for all use-cases of jMT.

¹⁰ The compilation scheme of [18] coincides with ours on common instructions.

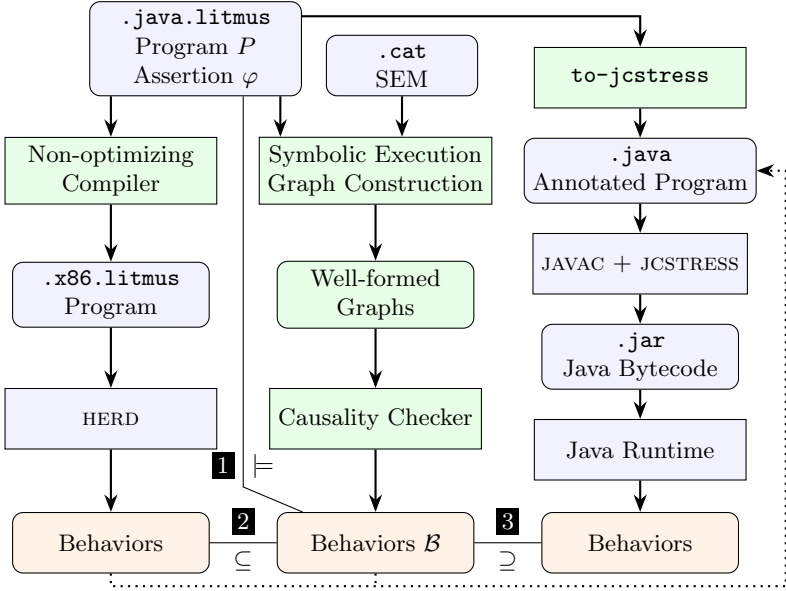


Fig. 4: Overview of jMT (components from others in blue, ours in green).

$\mathbf{b}$ in thread T_2 maps to $\mathbf{EAX}$ of T_2 . For registers appearing in behavior assertions, this mapping is bijective, hence invertible.

Note 1. One might notice that the behavior assertion should not matter here, but HERD in its listing of allowed behaviors only outputs the values of registers which appear in the behavior assertion. Therefore, we will only obtain and thus only need to invert mappings for registers that appear in the behavior assertion.

The soundness check of behavior inclusion as stated in (1) proceeds as follows:

1. **Compilation:** Source program P is compiled into an x86 program $Comp(P)$.
2. **Target hardware evaluation:** HERD enumerates all allowed behaviors $Beh_H(Comp(P))$ under the x86 memory model.
3. **Restoration:** The found behaviors are parsed, and the register mapping is inverted to express each behavior in terms of the original source registers.
4. **Inclusion check:** For each behavior $\beta \in Beh_H(Comp(P))$, jMT checks whether $\beta \in Beh_{MEM(SEM)}(P)$. Rather than full, upfront enumeration of β , jMT queries the source model on demand, avoiding exploration of unreachable behaviors that are costly due to MEM complexity.
5. **Verdict and witness:** If any behavior β fails the check, the tool reports *fail* and provides β as a witness. Otherwise, the test *passes*.

6 Tool jMT

Next, we shortly explain jMT's overall structure and usage.

Tool jMT. Figure 4 gives an overview of the toolchains of jMT. It depicts all three application scenarios. All scenarios start with a litmus test consisting of a program P and some assertion φ of required or forbidden behavior plus some single execution Java memory model SEM . Scenario 1 uses the behavior assertion; for scenarios 2 and 3 it is of no importance.

In all three scenarios, jMT builds the set of possible behaviors $\mathcal{B}^{11}$ of P under $MEM(SEM)$. In scenario 1, we check whether the required and forbidden behaviors as stated in φ are consistent with $\mathcal{B}$ (denoted $\models$). When applicable, jMT reports counterexamples.

Scenario 2 uses the non-optimizing compiler to x86 and the tool HERD as described in Section 5, and at the end compares the behaviors generated by HERD against $\mathcal{B}$. Again, counterexamples (of behaviors β allowed on x86 after compilation, but not contained in $\mathcal{B}$) are reported.

Finally, scenario 3 is similar to scenario 2, but now uses an (existing) optimizing compiler generating Java bytecode and during runtime optimized machine code. Here, we cannot use HERD anymore and instead have decided to employ the tool JCSTRESS [33]. JCSTRESS is a tool designed to aid in the validation of concurrency support for Java. Within our toolchain, we first generate such JCSTRESS annotated Java programs which are then compiled into bytecode. When executed, the program outputs a log with the observed behavior which can then again be compared against $\mathcal{B}$. Optionally (dotted line), our toolchain can directly feed either the behavior as generated by the non-optimizing compiler and HERD or $\mathcal{B}$ into the generation of the annotated program. In that case, checking behavior inclusion is directly part of the annotated program, making the test highly portable.

Usage. jMT is a command line tool. In its most basic form, jMT takes as input the program given in the HERD litmus test format for Java and a SEM in the .cat format of HERD. It will then check whether the behavior assertion of the litmus test is satisfied (scenario 1). Further options, e.g., to show the justification, are explained in jMT’s documentation. A number of programs can be used to execute parts of the toolchain of Figure 4.

- `to-herd-x86` generates the `.x86.litmus` programs employed in scenario 2.
- `with-herd-x86` builds on the former, but instead of displaying the generated x86 program it gives it to HERD to compute the allowed behaviors.
- `to-javasim` converts the program given in the HERD litmus test format for Java to the input format of the tool JAVASIM [21]. We need this conversion for our evaluation in Section 7.
- `with-javasim`: Similarly to `with-herd-x86`, this program uses JAVASIM to generate the allowed behaviors of the program.
- `to-jcstress` converts the program given in the HERD litmus test format for Java to a Java program ready for use with JCSTRESS.

¹¹ For optimization purposes, we actually do not construct *all* behaviors, but simply check inclusion element-wise.

Table 1: Test results of Java memory models (given as pass / fail / unsupported).

	# tests	jMT-JLS ₀₄	HERD-JAM ₂₁	JAVASIM
CTC 1–16	13	13 / 0 / 0	8 / 5 / 0	11 / 0 / 2
CTC 17, 18	2	2 / 0 / 0	2 / 0 / 0	0 / 2 / 0
jctestress - Java 5	27	27 / 0 / 0	27 / 0 / 0	27 / 0 / 0
jctestress - VarHandle	28	–	24 / 1 / 3	–
S&A	6	6 / 0 / 0	4 / 2 / 0	4 / 2 ¹² / 0
MT	24	24 / 0 / 0	16 / 6 / 2	18 / 4 / 2
total	100	72 / 0 / 0	81 / 14 / 5	60 / 8 / 4

7 Evaluation

Our evaluation of jMT is guided by the following research question:

RQ: Is jMT able to detect correctness issues in Java memory models?

More specifically, we are interested in the tool’s usefulness within our three application scenarios. To this end, we have chosen three Java memory models to test during the evaluation: The official Java language specification (JLS₀₄, [9], using jMT with the SEM from the extended paper); the model proposed in [18] (JAM₂₁, using HERD); and the multi-execution model implemented in JAVASIM [21] which aims to realize JLS₀₄. The programs (litmus tests) used during testing come from various sources: (1) CTC [28] are the well known “causality test cases” designed for evaluation of multi-execution models; (2) jctestress [33] are programs employed by JCSTRESS for testing Java implementations’ conformance to the OpenJDK developers understanding or opinion of Java 5 and **VarHandle** semantics; (3) S&A from [31]; and (4) MT from Manson’s thesis [20]. Tests corresponding to features outside our scope, such as final fields (unsupported in the Java syntax of HERD and orthogonal to **VarHandle**) and programs with loops are excluded. We converted all programs into the `.litmus` format of HERD and extended it with (required and forbidden) behavior assertions. A behavior is marked required (resp. forbidden) if expert documentation specifies it should be allowed (resp. forbidden), or if the Java language specification [9] applies (i.e., the program is in Java 5) and allows (resp. forbids) it. In addition, we include 69 further programs with no agreed-upon behavior. Interesting results on these programs thus need to be individually looked at, which we do for two such programs below.

Scenario 1. In this scenario, we check whether a proposed Java memory model allows all the required and forbids all the forbidden behaviors as described within a litmus test. In this case, we record it as “pass”; in all other cases (a required behavior is not allowed, or a forbidden behavior is possible) we record it as “fail”. Furthermore, we have outcome “unsupported” when tools do not support the input or crash on it. For example, HERD and JAVASIM do not support the

¹² For these incorrect results JAVASIM already gave a warning.

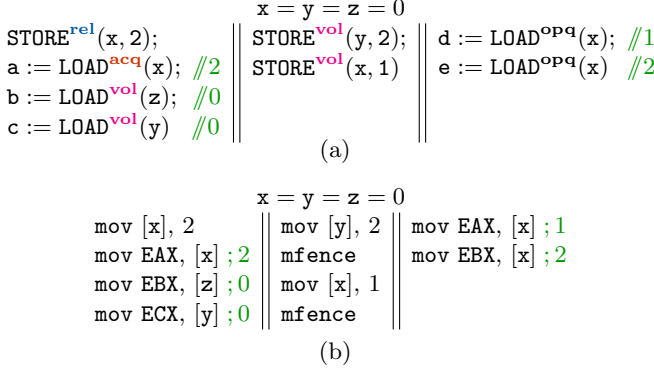


Fig. 5: A program (a) which when compiled to x86 (b) using the compilation scheme of [18] has a behavior (values in comments, `//` and `;`) that the source program does not allow according to their memory model JAM_{21} .

bitwise `or` operator and crash on programs containing it. We do not report timeouts, since all tests finish within a minute (most within a second, details in the extended paper). This performance is comparable to our experience with other weak-memory tools: fast enough for iterative model and test development.

Table 1 shows the summarized results of our experiments concerning scenario **1**. The full results are in the extended paper. We see that for two of the three proposed memory models we are able to find programs showing differences in the expected behavior and that of the memory model. For JLS_{04} and JAVASIM , we have elided experiments with litmus tests using the `VarHandle` API, because the language specification was developed prior to this API. An interesting result is that JAVASIM , designed to implement the Java language specification, actually differs from JLS_{04} on 8 litmus tests. Furthermore, we validated that `jMT`, given the JLS_{04} SEM, implements JLS_{04} .

Scenario 2. In these experiments, we check whether the behavior of a litmus test compiled to x86 with our non-optimizing compiler only shows behavior allowed by a memory model. On our test suite of scenario **1**, `jMT` finds no differences. We thus also ran our tool on the additional programs, revealing unsoundness of JAM_{21} with respect to compilation.

More specifically, one of the programs in our additional test set is taken from the JAM_{21} article (Section G.2.3.1 of the appendix of [18]). For this program, `HERD` (on the compiled program) calculates a behavior which JAM_{21} does not allow. This is slightly surprising since our non-optimizing compiler uses exactly the compilation scheme given in [18] for which the authors of [18] have proven correctness. We simplified the program to the core necessary to see the unsound compilation. The simplified program is given in Figure 5.

Obtaining such a result for a Java memory model might not necessarily mean that the memory model is broken, it could also be an unsound compilation scheme. Nevertheless, during the development of memory models such results are an indispensable aid for design which is what `jMT` is meant for.

Scenario 3. For this scenario, we use JCSTRESS and a standard Java compiler. Our experiments basically give the same results as for scenario 2: on our base test suite no differences are observable; on the extended set we find a number of programs with differing behavior.

One program with differences is again the one in Figure 5. Using our conversion to JCSTRESS, we were able to show that the result is also observed on x86 with OpenJDK 24 in about one in 18 thousand executions. Another interesting example where JAM₂₁ prevents a behavior that JCSTRESS can observe is the SB+rfis example from [15, Section 4, Remark 4]:

		$x = y = 0$		a b c d	samples	JAM ₂₁
STORE ^{pln} (x, 1);	$\left\ \begin{array}{l} \text{STORE}^{\text{pln}}(y, 1); \\ c := \text{LOAD}^{\text{vol}}(y); \text{//}1 \\ d := \text{LOAD}^{\text{vol}}(x) \text{//}0 \end{array} \right\ $	STORE ^{pln} (y, 1);	$\left\ \begin{array}{l} c := \text{LOAD}^{\text{vol}}(y); \text{//}1 \\ d := \text{LOAD}^{\text{vol}}(x) \text{//}0 \end{array} \right\ $	1 0 1 0	324 M	✗
a := LOAD ^{vol} (x); //1		c := LOAD ^{vol} (y); //1		1 0 1 1	1,240 M	✓
b := LOAD ^{vol} (y) //0		d := LOAD ^{vol} (x) //0		1 1 1 0	1,135 M	✓
				1 1 1 1	37 M	✓

On the left is the program; on the right the register values a–d observed by JCSTRESS. The result forbidden by JAM₂₁ occurs in 324 million samples. This is particularly interesting as it appears to be a counterexample to Lea’s explanation [16] of volatile semantics: “Volatile mode accesses are totally ordered.”

We interpret this statement as saying that the total order on volatile mode accesses is included in the antecedence relation defined in [16]. Otherwise, this total order would remain unnamed and play no role in the semantics. Under this interpretation, the forbidden outcome necessarily induces a coherence violation. Without loss of generality (the two threads are symmetric), assume that the volatile read $R_1^{\text{vol}}(x, a)$ precedes the volatile read $R_2^{\text{vol}}(y, c)$ in the total order on volatile accesses. Then we obtain the following cycle in the antecedence relation: $W_1^{\text{pln}}(x, 1) \xrightarrow{\text{antec.}} R_1^{\text{vol}}(x, a) \xrightarrow{\text{antec.}} R_2^{\text{vol}}(y, c) \xrightarrow{\text{antec.}} R_2^{\text{vol}}(x, d) \xrightarrow{\text{antidep.}} W_1^{\text{pln}}(x, 1)$. This cycle contradicts the acyclicity of antecedence, and therefore the outcome (a, b, c, d) = (1, 0, 1, 0) is forbidden under this reading of [16].

8 Conclusion

In this paper, we have introduced jMT, a tool for testing Java memory models against expected semantics and their compatibility with compilation schemes and compiler implementations. For its multi-execution model support, jMT builds on the concept of *symbolic execution graphs* which allows for an efficient representation of infinitely many potential candidates of execution graphs. The experimental evaluation shows that jMT can be a valuable aid in evaluating how JMMs match the mentioned desiderata. In the future, we intend to extend jMT with support for additional features, e.g., final fields and compilation to RISC-V. Additionally – as an alternative to the sampling approach of JCSTRESS – we could also statically analyze the machine code generated by the JIT compiler. Finally, we believe that jMT could play an important role in supporting the development of an updated JMM compatible with JLS₀₄ and the full `VarHandle` API.

Data availability statement. The models, tools, and scripts to reproduce our experimental evaluation are archived and available at [25].

References

1. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* **36**(2), 7:1–7:74 (2014). <https://doi.org/10.1145/2627752>
2. Aspinall, D., Ševčík, J.: Formalising Java’s data race free guarantee. In: Schneider, K., Brandt, J. (eds.) *TPHOL’07*. pp. 22–37. LNCS 4732, Springer (2007). https://doi.org/10.1007/978-3-540-74591-4_4
3. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: *ESOP. Lecture Notes in Computer Science*, vol. 9032, pp. 283–307. Springer (2015)
4. Bender, J., Palsberg, J.: A formalization of Java’s concurrent access modes. *Proc. ACM Program. Lang.* **3**(OOPSLA), 142:1–142:28 (2019)
5. Cenciarelli, P., Knapp, A., Sibilio, E.: The Java memory model: Operationally, denotationally, axiomatically. In: *ESOP. Lecture Notes in Computer Science*, vol. 4421, pp. 331–346. Springer (2007)
6. Gavrilenko, N., de León, H.P., Furbach, F., Heljanko, K., Meyer, R.: BMC for weak memory models: Relation analysis for compact SMT encodings. In: Dillig, I., Tasiran, S. (eds.) *CAV’19*. pp. 355–365. LNCS 11561, Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_19
7. Geeson, L., Smith, L.: Compiler testing with relaxed memory models. In: *CGO*. pp. 334–348. IEEE (2024)
8. Gosling, J., Joy, B., Steele, G.: *The Java language specification* (1996)
9. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., Smith, D., Bierman, G.: *Java language specification - Chapter 17. threads and locks - 17.4. memory model*, <https://docs.oracle.com/javase/specs/jls/se24/html/jls-17.html#jls-17.4>
10. Jeffrey, A., Riely, J., Batty, M., Cooksey, S., Kaysin, I., Podkopaev, A.: The leaky semicolon: compositional semantic dependencies for relaxed-memory concurrency. *Proc. ACM Program. Lang.* **6**(POPL), 1–30 (2022). <https://doi.org/10.1145/3498716>
11. Jorshari, M.H.K., Kokologiannakis, M., Majumdar, R., Nagendra, S.: Optimal concolic dynamic partial order reduction. In: Bouyer, P., van de Pol, J. (eds.) *CONCUR’25. LIPIcs*, vol. 348, pp. 26:1–26:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2025). <https://doi.org/10.4230/LIPICS.CONCUR.2025.26>
12. Kokologiannakis, M., Vafeiadis, V.: Genmc: A model checker for weak memory models. In: Silva, A., Leino, K.R.M. (eds.) *CAV’21*. pp. 427–440. LNCS 12759, Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_20
13. Lahav, O.: A case against semantic dependencies. In: *The Future of Weak Memory 2024*. <https://popl24.sigplan.org/details/fowm-2024-papers/8/A-case-against-semantic-dependencies>, recording of talk available at <https://youtu.be/7VZtwIvZ6qE>
14. Lahav, O., Boker, U.: Decidable verification under a causally consistent shared memory. In: Donaldson, A.F., Torlak, E. (eds.) *PLDI’20*. pp. 211–226. ACM (2020). <https://doi.org/10.1145/3385412.3385966>
15. Lahav, O., Vafeiadis, V., Kang, J., Hur, C., Dreyer, D.: Repairing sequential consistency in C/C++11. In: Cohen, A., Vechev, M.T. (eds.) *PLDI’17*. pp. 618–632. ACM (2017). <https://doi.org/10.1145/3062341.3062352>
16. Lea, D.: Using JDK 9 memory order modes, <https://gee.cs.oswego.edu/dl/html/j9mm.html>

17. Liu, S., Bender, J., Palsberg, J.: Compiling volatile correctly in Java. In: ECOOP. LIPIcs, vol. 222, pp. 6:1–6:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
18. Liu, S., Bender, J., Palsberg, J.: Compiling volatile correctly in Java. In: Ali, K., Vitek, J. (eds.) ECOOP’22. pp. 6:1–26. LIPIcs 222, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). <https://doi.org/10.4230/LIPICS.ECOOP.2022.6>
19. Manson, J.: Volatile fields and synchronization, <https://jeremymanson.blogspot.com/2007/03/volatile-fields-and-synchronization.html>
20. Manson, J.: The Java Memory Model. Ph.D. thesis, University of Maryland, College Park, MD, USA (2004), <https://hdl.handle.net/1903/1949>
21. Manson, J., Pugh, W.: The Java memory model simulator. In: FTfJP’02 (2002)
22. Manson, J., Pugh, W., Adve, S.: The Java memory model <https://web.archive.org/web/20060303190317/http://www.cs.umd.edu/users/jmanson/java/journal.pdf>, original URL: <http://www.cs.umd.edu/users/jmanson/java/journal.pdf>, archived 2006-03-03
23. Manson, J., Pugh, W.W., Adve, S.V.: The Java memory model. In: POPL. pp. 378–391. ACM (2005)
24. Moiseenko, E., Kokologiannakis, M., Vafeiadis, V.: Model checking for a multi-execution memory model. Proc. ACM Program. Lang. **6**(OOPSLA2), 758–785 (2022). <https://doi.org/10.1145/3563315>
25. Panneke, L., Wehrheim, H.: jMT: Testing correctness of Java memory models (artifact) (Oct 2025). <https://doi.org/10.5281/zenodo.17634679>
26. Paviotti, M., Cooksey, S., Paradis, A., Wright, D., Owens, S., Batty, M.: Modular relaxed dependencies in weak memory concurrency. In: Müller, P. (ed.) ESOP20. pp. 599–625. LNCS 12075, Springer (2020). https://doi.org/10.1007/978-3-030-44914-8_22
27. Petri, G., Vitek, J., Jagannathan, S.: Cooking the books: Formalizing JMM implementation recipes. In: Boyland, J.T. (ed.) ECOOP’15. LIPIcs, vol. 37, pp. 445–469. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). <https://doi.org/10.4230/LIPICS.ECOOP.2015.445>
28. Pugh, W.: Causality test cases, <https://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html>
29. Pugh, W.W.: Fixing the Java memory model. In: Java Grande. pp. 89–98. ACM (1999)
30. Richards, J., Wright, D., Cooksey, S., Batty, M.: Symbolic MRD: Dynamic memory, undefined behaviour, and extrinsic choice. Proc. ACM Program. Lang. **9**(OOPSLA1), 1858–1882 (2025). <https://doi.org/10.1145/3721089>
31. Ševčík, J., Aspinall, D.: On validity of program transformations in the Java memory model. In: Vitek, J. (ed.) ECOOP’08. pp. 27–51. LNCS 5142, Springer (2008). https://doi.org/10.1007/978-3-540-70592-5_3
32. Shipilëv, A.: Close encounters of the Java memory model kind, <https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/>
33. Shipilëv, A., et al.: Java concurrency stress (jctestress) (2013), <https://github.com/openjdk/jctestress/>
34. Trippel, C., Manerkar, Y.A., Lustig, D., Pellauer, M., Martonosi, M.: TriCheck: Memory model verification at the trisection of software, hardware, and ISA. In: Chen, Y., Temam, O., Carter, J. (eds.) ASPLOS’17. pp. 119–133. ACM (2017). <https://doi.org/10.1145/3037697.3037719>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Demonstrating *ARG-V*'s Generation of Realistic Java Benchmarks for SV-COMP

Charles Moloney¹ , Robert Dyer¹ , and
Elena Sherman²

¹ University of Nebraska–Lincoln, Nebraska, USA
`{cmoloney2@huskers.unl,rdyer@unl}.edu`

² Boise State University, Idaho, USA
`elenasherman@boisestate.edu`

Abstract. The SV-COMP competition provides a state-of-the-art platform for evaluating software verification tools on a standardized set of verification tasks. Consequently, verifier development outcomes are influenced by the composition of program benchmarks included in SV-COMP. When expanding this benchmark corpus, it is crucial to consider whether newly added programs cause verifiers to exhibit behavior distinct from that observed on existing benchmarks. Doing so helps mitigate external threats to the validity of the competition's results.

In this paper, we present the application of the *ARG-V* tool for automatically generating Java verification benchmarks in the SV-COMP format. We demonstrate that, on a newly generated set of 68 realistic benchmarks, all four leading Java verifiers decrease in accuracy and recall compared to their performance on the existing benchmark suite. These findings highlight the potential of *ARG-V* to enhance the comprehensiveness and realism of verification tool evaluation, while also providing a roadmap for verifier developers aiming to improve their tools' applicability to real-world software.

Keywords: formal verification · benchmark synthesis · open-source research tool

1 Introduction

One of the persistent challenges in developing software verifiers is a lack of publicly available and representative benchmark programs for evaluating functionality and identifying new edge cases to be solved [9]. To address this problem, researchers have come together to share resources and provide a common platform for benchmarking in groups like SV-COMP [1]. These have been a massive boon to the formal verification community, but it is important to be aware of how the nature of these benchmarks can influence competition results and, ultimately, verifier development. To be considered as valid Java SV-COMP benchmarks, programs must only leverage the Java Class Library, inject a shared “Verifier” library, and have an associated property key for evaluation. Generation of a

realistic (i.e., mined from real-world code) SV-COMP Java benchmark would typically consist of:

1. Identification of code with desirable characteristics, such as conditional statements or arithmetic expressions.
2. Removal of non-JDK dependencies and irrelevant code fragments.
3. Instrumenting programs with a common “Verifier” variable(s).
4. Creation of verification scaffolding files with expected verdicts.
5. Definition of properties for verification tasks and modification of code to target them.

Performing each step manually is labor-intensive and requires human intervention, which might introduce bias; benchmark selection alone can lead to a disproportionate number of benchmarks exhibiting characteristics that verifiers can already efficiently handle. Impartial selection would reduce the number of opportunities for bias and increase the likelihood of new benchmarks exploring underdeveloped features for existing verifiers. In an effort to mitigate this bias and enable developers to skip directly to the final step of the generation process, we utilize *ARG-V* [18], our tool for automating the time-consuming filtering and formatting necessary to get a benchmark into a required format.

This paper focuses on the application of *ARG-V* to meaningfully augment the SV-COMP corpus. It explores how the *ARG-V* generated benchmarks increase the representation of program behaviors, which reduces threats to the external validity of SV-COMP results. We have generated a new set of benchmarks for consideration at SV-COMP and evaluated on them the behavior of four prominent Java verifiers. At the time of this paper, 70 *ARG-V* benchmarks have been merged into the SV-COMP Java library across the categories of reachability safety and runtime exception safety. By leveraging the impartiality and filter functionality of our tool, we hope to assist the formal verification community by accelerating the production of more benchmarks, pushing the boundaries and coverage to new areas in existing public corpora.

2 *ARG-V* Tool Description

ARG-V builds on a previous tool, *PAClab: A Program Analysis ColLABoratory* [4], by expanding its capabilities and offering the option to perform transformations targeted to the SV-COMP format. While *PAClab* is designed to create benchmarks on the intra-procedural level, transforming methods using symbolic booleans and integers for Java Symbolic PathFinder [15], *ARG-V* substantially extends *PAClab*’s scope to handle artifact preparation, additional data types, field variables, and cross-class interactions. Table 1 highlights the major changes between the tools.

ARG-V’s architecture consists of three distinct modules:

1. **Download**, which fetches a number of GitHub repositories as specified by a CSV input file.

Table 1. Comparison showing the expanded functionality of *ARG-V*

Feature	PAClab	ARG-V
Scope	Intra-procedural (Method-level only)	Inter-procedural (Class-level and JDK packages)
Member Variables	Static only	Static and Non-Static
Supported types	Only ints and booleans	All primitive types and primitive arrays
Type Resolution	Implicit resolution only	Implicit and Explicit resolution via method bindings in classpath and type table
Artifact Preparation	None	Creation of main method and benchmark config (yaml) file, preservation of provenance information, and re-naming to formatting standard

2. **Filter**, which narrows down the repository files to those containing user-set minimums of specified features, such as if statements, type expressions, type parameters, or conditionals.
3. **Transform**, which breaks the code down into an abstract syntax tree (AST) and performs a series of modifying operations, preparing the file for use as a benchmark and generating any necessary artifacts

The transform module leverages the Eclipse Foundation’s JDT [7] for AST traversals, using a type table and method bindings to make informed decisions about code removal and replacement (a more detailed explanation is described in the original *PAClab* paper [4]). This allows the code to compile independently of its package or source repository without losing all meaning, strategically removing external dependencies while keeping as much computational logic as possible.

3 Generation and Testing of New Benchmarks

To demonstrate the efficacy of *ARG-V* as a benchmark generator, we use it to generate 68 new benchmarks to augment the SV-COMP Java corpus [10]. These benchmarks encompass two categories: **ReachSafety**, which resolves to true if all assertions hold in the code, and **ExceptionProperty**, which resolves to true if the code cannot throw a `RuntimeException`. Some of these benchmarks contain noteworthy characteristics for both categories, so the resulting set consists of 48 **ReachSafety** benchmarks (31 true, 17 false) and 50 **ExceptionProperty** benchmarks (39 true, 11 false) – 98 property runs in total.

For the generation of these benchmarks, *ARG-V* scrapes the first 1,261 repositories within the RepoReaper database [13], an open-source curator of publicly available GitHub repositories. We set the criteria to require at least one ‘if’ statement that is conditional on a chosen primitive type (floating point or integer) to ensure multiple paths through the code. This minimal criterion guarantees that files without interesting logic (e.g., only getters and setters) will not be downloaded. We avoid stricter criteria that would miss useful benchmarks, as they result in a less impartial result set by concentrating the resulting corpus around

a very specific criterion, such as 3 or more branching conditions. Assert statements are added in manually, checking for variable value ranges or null-safety. During any manual modifications, an emphasis is put on basic constructs, primitive values, and arrays, with no recursion³ or complex data structures. The final benchmarks contain an average of 41.7 lines of code per file, comparable to the average 42.2 lines of code in existing SV-COMP Java benchmarks [19].

The main goal of this tool demonstration is to assess whether *ARG-V* is capable of producing a distinct corpus of benchmarks that provide a fresh challenge for Java verifiers, as well as whether there is a significant difference in difficulty between the realistic *ARG-V* benchmarks and the existing SV-COMP corpus. As a representation of the state of the art in Java verification, the new benchmarks are evaluated using the four top-performing Java verifiers from SV-COMP 2025 [1]: MLB [5, 14], GDart [11, 12], JavaRanger [17, 8], and JBMC [3, 16].

To standardize our evaluation, experiments are run using SoSy-Lab’s BenchExec [2], the same platform used in the SV-COMP competition. Each tool is executed on a verifier run and limited to 4GB of memory, 120 seconds of cumulative CPU time, and 2 processing units. The benchmarking machine is an Intel Core i9-12900HK CPU at 2.5GHz, 32 GB of RAM, running Ubuntu 24.04.1 LTS with BenchExec version 3.30 [20]. All benchmarks are run using the same parameters; the complete command for executing JBMC can be seen below:

```
scripts/execute_runs/execute-runcollection.sh \
benchexec/bin/benchexec archives/2025/jbmc.zip \
benchmark-defs/jbmc.xml witness.graphml results-verified/ \
-t ReachSafety-Java -t RuntimeException-Java \
--timelimit 120 --memorylimit 4GB --limitCores 2
--numOfThreads 8 --read-only-dir / --overlay-dir /home/ \
--overlay-dir ./ --tool-directory . -o .graphml
```

4 Evaluation of *ARG-V* Benchmarks against SV-COMP

The same verifiers are also run with the same parameters on the existing SV-COMP corpus of 674 Java benchmarks, which contains of 674 *ReachSafety* properties and 673 *ExceptionProperty* properties. We analyze the differences in verification results between these two benchmark sets. To avoid overlap, the SV-COMP benchmarks produced by *ARG-V* are omitted from this corpus.

We compare the results of the verifiers on the two benchmark corpora using standard classification metrics: **accuracy**, the proportion of correctly classified samples; **precision**, the fraction of predicted positives that are actually true positives; **recall**, the fraction of actual positives correctly identified versus the

³ From our experience, we find that recursion often leads to timeouts or unknown verdicts from verifiers. The decision was made to avoid recursive benchmarks when generating our sample corpus, as it might not lead to meaningful results from the verifiers on our new benchmarks.

Table 2. Classification metrics between benchmark sets, excluding “unknown”, “error”, and “timeout”

	reach-safety		runtime-exception		cumulative	
	existing	new	existing	new	existing	new
Accuracy	1.00	0.72	0.67	0.72	0.83	0.72
Precision	0.99	0.96	1.00	0.89	0.99	0.90
Recall	1.00	0.49	0.66	0.73	0.75	0.64
Specificity	0.99	0.98	0.91	0.68	0.99	0.88

total number of actual positives; and **specificity**, the fraction of actual negatives that were correctly identified compared to the total number of actual negatives.

These values are calculated for both categories, as well as the cumulative score across **ReachSafety** and **ExceptionProperty**. Aggregate results from all verifiers are available in Table 2; complete results and raw data are available in the replication package [10]. Results for Table 2 are calculated strictly on affirmative outputs, thus omitting undecidable results, i.e., when a verifier returns “unknown”, “error”, or “timeout” rather than “true” or “false”.

Across all reach-safety and cumulative metrics, we find that verifiers’ performances **decrease substantially** on the new, realistic benchmarks compared to the existing SV-COMP corpus. When evaluating if a program will ever have false assertions, the verifiers are less than half as likely to make a correct decision. The verifiers do perform relatively consistently between the new and existing benchmarks within the runtime-exception category, but only when undecidable results are omitted from the results.

SV-COMP also allows verifiers to return an “unknown” result when they cannot solve a verification task [1]. It is relevant to analyze the performance on these benchmarks while including this uncertainty option, as a higher inability to make a judgment on a new set of benchmarks suggests potential categories of benchmarks that the verifier is not well-equipped to handle. Table 3 shows the same classification metrics as before, but with undecidable metrics counting as a failed result. Precision is omitted, as there is no unknown-inclusive analog for this metric.

With undecidable results included, the verifiers have a **notably poorer performance** for all metrics across both checked properties. The runtime-exception property, which verifiers exhibit comparable performance on for unknown-exclusive data, has a much larger disparity in Table 3. This is due to a substantially higher proportion of undecidable responses on the new benchmarks.

These results suggest a significant difference in verifier capability between the two corpora. A cumulative recall of only 0.271 for the new benchmarks versus the 0.554 of the SV-COMP corpus means verifiers are less than half as likely to be able to correctly identify a property as true on a similarly-scoped benchmark when it does not come from the SV-COMP suite. A substantial drop in specificity denotes a similar inability to correctly identify a property as false, resulting in an overall much lower accuracy on new benchmarks.

Table 3. Undecidable-Inclusive metrics with “unknown”, “error”, or “timeout” counting as a failed result. Precision is unaffected by this change.

	reach-safety		runtime-exception		cumulative	
	existing	new	existing	new	existing	new
Accuracy	0.75	0.33	0.49	0.35	0.62	0.34
Recall	0.71	0.18	0.49	0.35	0.55	0.27
Specificity	0.76	0.60	0.54	0.34	0.75	0.50
% Undecidable	25%	55%	26%	52%	26%	53%

Table 4. Results of individual verifiers across all metrics. Undecidable Inclusive (UI) metrics include “unknown”, “error”, or “timeout” as a failed result.

	java-ranger		mlb		gdart		jbmc	
	existing	new	existing	new	existing	new	existing	new
Accuracy	0.70	0.67	0.66	0.51	1.00	0.86	0.99	0.83
Precision	0.99	1.00	0.98	0.74	1.00	1.00	1.00	0.95
Recall	0.57	0.50	0.49	0.43	1.00	0.75	0.99	0.78
Specificity	0.99	1.00	0.99	0.68	1.00	1.00	1.00	0.92
UI-Accuracy	0.48	0.08	0.56	0.31	0.63	0.32	0.81	0.64
UI-Recall	0.40	0.06	0.39	0.24	0.59	0.21	0.84	0.57
UI-Specificity	0.65	0.14	0.91	0.46	0.71	0.57	0.76	0.82
% Undecidable	32%	88%	18%	22%	37%	63%	18%	22%

Table 4 breaks the results out across individual verifiers. For instance, results for GDart [11, 12] indicate it is possible the model is overfitting on the current SV-COMP corpus, given its much lower performance and higher unknown/timeout/error rate on new benchmarks of similar complexity.

The ability of *ARG-V* to naïvely (i.e., no deliberate attempt to fool verifiers or intentional selection of arbitrarily complex code) produce benchmarks that are difficult for the current state-of-the-art verifiers is promising and shows the importance of a richer set of benchmark programs obtained from open-source code. We intend for this tool to be used to extend the existing boundaries of software verifiers further, and doing so will require benchmark suites that are more comprehensive and explore new, unaddressed areas.

Leveraging the parameterized filter step, users can further test for specific features of interest. For instance, many of the individual benchmarks that see a high failure rate within the *ARG-V* sample corpus involve multiple branching conditions operating on nondeterministic floating point values [10]; a researcher hoping to test and identify more failing edge cases for this property can filter for benchmark candidates, using parameters such as a high `minIfStmt` threshold, and select methods using symbolic `double` and `float` types.

5 Conclusion and Future Work

We present *ARG-V*, an Adaptable Realistic Benchmark Generator for Verification. Using this tool, we generate a sample corpus of 68 benchmarks for use in the SV-COMP Java benchmark library, and these benchmarks are capable of

providing a substantial increase in challenge for the state of the art in software verification, suggesting unknown blind spots in verifier logic and/or overfitting to public benchmark repositories. We believe this tool provides an efficient and streamlined method for generating new benchmarks at scale, illuminating new unaddressed problem spaces, and identifying more edge cases for verifiers. The underlying source(s) for the higher difficulty solving tests generated by *ARG-V* remains a largely open question. We can confidently say it is not caused by the length of the code (as the *ARG-V* generated benchmarks are shorter), the use of floating-point operations (as verifiers do well on prior benchmarks with floating-point), or the use of Java Standard Library functions. We hope to use this set of benchmarks as a starting point to investigate this specific problem. Additionally, we will continue to work to make development for verifier researchers faster and more comprehensive, and we plan to continue to create more benchmarks using *ARG-V* as we add new features. This tool can still be expanded in a variety of ways: we are actively working to add more granular filtering methods, supporting the transformation of more data structures and edge cases within the Java library, and exploring leveraging deep learning to prevent creating benchmarks already covered by similar tests within a corpus. Development is also underway for an analogous C-code transformer [6].

Acknowledgments. We thank Jon Craig and Rubal Goyal for helping with an earlier implementation of the tool and benchmarks. This tool was partially funded by the U.S. National Science Foundation under grants 1823294, 1823357, 2346326, and 2346327.

Data Availability

The source code and build steps are available in the replication package [10]. For reproduction, use version 1.0.1, and for general reuse, use the most recent version available.

References

1. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS (3). pp. 151–186. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_9
2. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (Feb 2019). <https://doi.org/10.1007/s10009-017-0469-y>, <https://doi.org/10.1007/s10009-017-0469-y>
3. Brenguier, R., Cordeiro, L., Kroening, D., Schrammel, P.: JBMC: A bounded model checking tool for Java bytecode (2023), <https://arxiv.org/abs/2302.02381>
4. Brunner, R., Dyer, R., Paquin, M., Sherman, E.: PAClab: a program analysis col-laboratory. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1616–1620. ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA (2020). <https://doi.org/10.1145/3368089.3417936>

5. Bu, L., Liang, Y., Xie, Z., Qian, H., Hu, Y.Q., Yu, Y., Chen, X., Li, X.: Machine learning steered symbolic execution framework for complex software code. *Form. Asp. Comput.* **33**(3), 301–323 (Jun 2021). <https://doi.org/10.1007/s00165-021-00538-3>, <https://doi.org/10.1007/s00165-021-00538-3>
6. Craig, J., Sherman, E., Dyer, R.: GitHub - unl-pal/argv-c-transformer — github.com. <https://github.com/unl-pal/argv-c-transformer> (2025), [Accessed 13-10-2025]
7. Eclipse Foundation: Eclipse JDT (Java Development Tools) (2024), <https://projects.eclipse.org/projects/eclipse.jdt>
8. Hussein, S., Kim, K., McCamant, S., Sharma, V., Whalen, M.: Java Ranger - SV-COMP 2025 (2024). <https://doi.org/10.5281/zenodo.14205198>
9. Kalibera, T., Parizek, P., Haddad, G., Leavens, G.T., Vitek, J.: Challenge benchmarks for verification of real-time programs. In: *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages Meets Program Verification*. p. 57–62. PLPV '10, Association for Computing Machinery, New York, NY, USA (2010). <https://doi.org/10.1145/1707790.1707800>, <https://doi.org/10.1145/1707790.1707800>
10. Moloney, C., Dyer, R., Sherman, E.: Replication package for “Demonstrating ARG-V’s Generation of Realistic Java Benchmarks for SV-COMP” (2025). <https://doi.org/10.5281/zenodo.17235880>
11. Mues, M., Howar, F.: GDart: An ensemble of tools for dynamic symbolic execution on the java virtual machine (competition contribution). In: *Proc. TACAS* (2). p. 435–439. Springer-Verlag, Berlin, Heidelberg (2022). https://doi.org/10.1007/978-3-030-99527-0_27
12. Mues, M., Howar, F.: GDart SV-COMP 2025 (2024). <https://doi.org/10.5281/zenodo.14207413>
13. Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M.: Curating GitHub for engineered software projects. *Empirical Softw. Engg.* **22**(6), 3219–3253 (Dec 2017). <https://doi.org/10.1007/s10664-017-9512-6>, <https://doi.org/10.1007/s10664-017-9512-6>
14. Peiyang, L.: MLB SV-COMP 2024 (2024). <https://doi.org/10.5281/zenodo.14229077>
15. Păsăreanu, C.S., Rungta, N.: Symbolic PathFinder: symbolic execution of Java bytecode. In: *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. p. 179–180. ASE '10, Association for Computing Machinery, New York, NY, USA (2010). <https://doi.org/10.1145/1858996.1859035>, <https://doi.org/10.1145/1858996.1859035>
16. Schrammel, P.: JBMC: Bounded model checking for Java bytecode (2024). <https://doi.org/https://doi.org/10.5281/zenodo.14207455>
17. Sharma, V., Hussein, S., Whalen, M.W., McCamant, S., Visser, W.: Java Ranger: statically summarizing regions for efficient symbolic execution of Java. In: *FSE 2020 Proceedings*. p. 123–134. ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA (2020). <https://doi.org/10.1145/3368089.3409734>, <https://doi.org/10.1145/3368089.3409734>
18. Sherman, E., Dyer, R., Craig, J., Moloney, C., Flint, S.W., Goyal, R., Brunner, R., Paquin, M.: ARG-V - Adaptable Realistic Benchmark Generator for Verification (2025), <https://arg-v.dev/>
19. SoSy-Lab: SV-Benchmarks: Benchmark Set for SV-COMP 2025 (tag `svcomp25`). <https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tags/svcomp25> (2025), final benchmark set for SV-COMP 2025, commit `1e5856db` (released Mar. 17, 2025)

20. Wendler, P., Beyer, D.: sosy-lab/benchexec: Release 3.30 (2025). <https://doi.org/10.5281/zenodo.15606624>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Hardware Verification and Model Checking



CTL* Model Checking on Infinite Families of Finite-State Labeled Transition Systems

Roberto Pettinau¹ and Christoph Matheja^{1,2}

¹ Carl von Ossietzky University of Oldenburg, Oldenburg, Germany
{roberto.pettinau,christoph.matheja}@uol.de

² DTU Compute, Kongens Lyngby, Denmark

Abstract. We study model checking algorithms for infinite families of finite-state labeled transition systems against temporal properties written in CTL*. Such families arise, for example, as models of highly configurable systems or software product lines. We model families using context-free graph grammars. We then develop a state labeling algorithm that works compositionally on the grammar’s production rules with limited information about the context in which the rule is applied. The result is a graph grammar modeling the same family but with extended labels. We leverage this grammar to decide whether all, some, or (in)finitely many members of a family satisfy a given temporal property. We have implemented our algorithms and present early experiments.

1 Introduction

Modern computer systems are increasingly built as highly-configurable product families that vary by features or performance parameters (cf. [41,4,6]). Such systems are inherently complex, because numerous feature combinations may result in product configurations that are deployed – and potentially fail – for the first time. Hence, techniques to ensure the correctness of entire product families are crucial. One such technique is model checking, which has been successfully applied to certain classes of families, such as software product lines [12,6,11]. Existing model-checking approaches often reason about families of structurally similar systems that are derived from a finite-state super-system, where all features are enabled [11]. However, complex product families may not admit such an all-in-one super-system. For example, if the number of some components is configurable, one needs to consider infinite families of structurally evolving systems. Similar families arise in distributed systems with configurable network topologies [24].

In this paper, we develop model checking algorithms for infinite families $\{T_i\}_{i \in \mathbb{N}}$ of labeled transition systems (LTSs) that can evolve in size and structure. While every LTS T_i is finite-state, it can be arbitrarily large. To represent such families, we use hyperedge replacement grammars (HRGs), a well-studied extension of context-free grammars to describe sets of graphs [16,21]. Figure 1a depicts an HRG modeling a family of colored doubly-linked lists (see Figure 1c),

which will serve us as a running toy example. The boxes are hyperedges. They correspond to nonterminals that are replaced by graphs according to the grammar's rules, see Figure 1b for an illustration. Node colors indicate atomic propositions. Our algorithms are based on a solution to the *recoloring problem*, which informally reads as follows: Given a family F modeled by an HRG $\mathcal{G}$ and an ω -regular property φ , compute a *recolored* HRG $\mathcal{G}_\varphi$ modeling the same family F with one exception: For every $T_i \in F$ and every state s of T_i with $(T_i, s) \models \varphi$, the state s is additionally colored with φ .

Being able to compute recolored HRGs $\mathcal{G}_\varphi$ recursively yields decision procedures for various verification questions, including:

- Does every (or some) LTS in the family F satisfy the ω -regular property φ ?
- Are there finitely (or infinitely) many LTSs in F that satisfy φ ?
- Given formula Ψ in CTL* [20], does every (or some) LTS in F satisfy Ψ ?
- Given a formula ψ in the qualitative fragment of the probabilistic branching time logic PCTL [26][5, p. 787], does every *Markov chain* in F satisfy ψ ?

To illustrate how we compute recolored HRGs, consider the LTS in Figure 1c, which is a member of the family specified by our example grammar in Figure 1a. As indicated by the rectangles, the LTS can be decomposed into three parts: (i) an application of the production rule $\mathcal{R}_2$ (green box), (ii) the graph $\mathcal{K}$ plugged into the nonterminal of $\mathcal{R}_2$, and (iii) the context $\mathcal{J}$ in which the rule has been applied. To determine whether the node v of $\mathcal{R}_2$ should be colored with an ω -regular property, say **red** U **blue**, we do not need full knowledge about $\mathcal{J}$ and $\mathcal{K}$. It suffices to know that (1) we can reach a **blue** node from v by taking an edge from $\mathcal{R}_2$ into $\mathcal{J}$ via $\mathcal{K}$ and that (2) we can globally stay in **red** nodes by taking edges from $\mathcal{R}_2$ into $\mathcal{J}$ or $\mathcal{K}$. Due to (2), **red** U **blue** is not satisfied for all traces starting in v . Hence v should *not* be colored with **red** U **blue**. More generally, we derive an equivalence relation over graphs from any ω -regular property that groups the possible graphs $\mathcal{J}$ and $\mathcal{K}$ composed with the rule $\mathcal{R}_2$ into classes that represent observations like (1) and (2). We then show that those equivalence relations form a congruence and that it suffices to consider only finitely many equivalence classes. Hence, we can solve the recoloring problem through a compositional analysis of the grammar's production rules. The same approach can be applied recursively to reason about CTL* instead of ω -regular properties.

Contributions. Our main contributions can be summarized as follows:

- We develop an algorithm for the novel *recoloring problem* for infinite families of finite-state labeled transition systems modeled by graph grammars.
- We prove our algorithm for the recoloring problem sound and complete.
- We extend our solution to the recoloring problem to decide whether all, some, or (in)finitely many members of an LTS family satisfy a CTL* formula.
- We have implemented our algorithms. Experiments demonstrate that properties in CTL* and in the qualitative fragment of PCTL can be verified within seconds for simple families of finite-state systems.

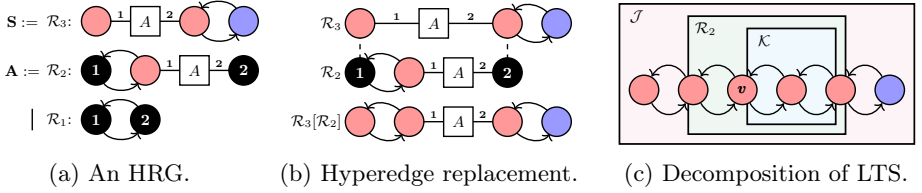


Fig. 1: Toy example of a family of LTSs modeled by an HRG. Each LTS in the family is a doubly linked list where all nodes but the rightmost one are red.

Outline. Section 2 introduces preliminaries and formally states the recoloring problem. Section 3 presents a general approach for compositional reasoning about HRGs that we apply in Section 4 to the recoloring problem. Section 5 develops the resulting algorithm for recoloring and extends it to CTL*. Section 6 reports on a prototypical implementation and experiments. Finally, Section 7 discusses related work and concludes. Proofs and further details about our experiments are available online in a technical report [35].

2 Preliminaries and Formal Problem Statement

Our model checking technique for families is based on three ingredients that we recap first: Büchi automata, hypergraphs, and hyperedge replacement grammars.

Basic notation. X^* (X^ω) is the set of all finite (infinite) sequences $\bar{x} = x_1x_2x_3 \dots$ over the set X . The (possibly infinite) length of $\bar{x}$ is $|\bar{x}|$. The empty sequence is ε . We denote by $\bar{x}|_Y$ the sequence obtained by keeping in $\bar{x}$ only elements in the set Y . The concatenation of $\bar{x}$ and $\bar{y}$ is $\bar{x}\bar{y}$. We denote by $X \uplus Y$ the union of the sets X and Y if $X \cap Y = \emptyset$; otherwise, $X \uplus Y$ is undefined. For functions $f: X \rightarrow A$ and $g: Y \rightarrow B$, the function $f \uplus g: (X \uplus Y) \rightarrow (A \cup B)$ maps every $x \in X$ to $f(x)$ and every $y \in Y$ to $g(y)$. We lift functions to sequences and sets by pointwise application, e.g. $f(x_1x_2 \dots) = f(x_1)f(x_2) \dots$ and $f(Y) = \{f(y) \mid y \in Y\}$.

2.1 Büchi Automata

CTL* model checking (cf. [5, p. 429]) based on recursively labeling states with satisfaction sets turns any LTL model checking algorithm into a CTL* model checker. We thus first study model checking of infinite families for ω -regular properties specified by Büchi automata, which subsumes LTL (cf. [40,5]).

Definition 1. A Büchi automaton is a tuple $\mathcal{M} = (Q, \Sigma, \Delta, Q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation, $Q_0 \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of final states. We denote by $\text{BUCHI}(\Sigma)$ the set of all Büchi automata over the alphabet Σ .

A Büchi automaton $\mathcal{M} = (Q, \Sigma, \Delta, Q_0, F)$ accepts an infinite word $\bar{\sigma} \in \Sigma^\omega$ if it admits a run that visits a final state infinitely often. We define both finite and infinite runs as our model checking algorithm uses them for abstraction. A *finite* (resp. *infinite*) *run* of $\mathcal{M}$ is a sequence $\bar{r} = q_0\sigma_0 \dots \sigma_{n-1}q_n$ (resp.

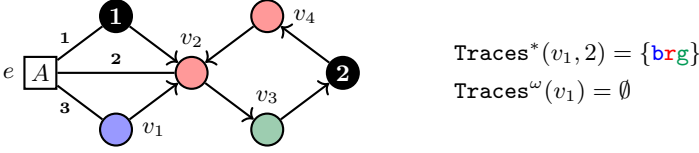


Fig. 2: Illustration of a hypergraph together with two sets of traces.

$\bar{r} = q_0\sigma_0q_1\sigma_1\cdots$) such that $(q_i, \sigma_i, q_{i+1}) \in \Delta$ for all $0 \leq i < n$ (resp. $i \geq 0$). We denote by Runs^* and Runs^ω the sets of all finite and infinite runs of $\mathcal{M}$, respectively. $\text{inf}(\bar{r})$ denotes the set of states that appear *infinitely often* in run $\bar{r}$. We extend the transition relation Δ to take finite (infinite) words and indicate with a flag $b \in \{\mathbf{F}, \mathbf{T}\}$ whether some final states have been visited (infinitely often):

$$\begin{aligned}\Delta^* &= \{(q_0, \bar{r}|_\Sigma, q_n, b) \mid \bar{r} \in \text{Runs}^*, q_0 = \text{first}(\bar{r}), q_n = \text{last}(\bar{r}), b = (\bar{r}|_F \neq \varepsilon)\} \\ \Delta^\omega &= \{(q_0, \bar{r}|_\Sigma, b) \mid \bar{r} \in \text{Runs}^\omega, q_0 = \text{first}(\bar{r}), b = (\text{inf}(\bar{r}) \cap F \neq \emptyset)\}\end{aligned}$$

The language of $\mathcal{M}$ is defined as $\mathcal{L}(\mathcal{M}) = \{\bar{\sigma} \in \Sigma^\omega \mid \exists q_0 \in Q_0. (q_0, \bar{\sigma}, \mathbf{T}) \in \Delta^\omega\}$.

2.2 Hypergraphs

Hypergraphs (HG) generalize ordinary graphs by allowing *hyperedges* to connect an arbitrary number of nodes. To simplify constructions, we distinguish between ordinary edges and hyperedges. From now on, we fix a set **COL** of node *colors* (think: atomic propositions), a set **ACT** of *actions* (think: ordinary edge labels), and a set **LAB** of *hyperedge labels*. Moreover, we equip HGs with a fixed number of uncolored abstract nodes that will be used for gluing HGs together.

Definition 2. A hypergraph is a tuple $\mathcal{H} = (V, n, \rightarrow, E, \alpha, \gamma, \ell)$ whose set of nodes $W = V \uplus I$ is partitioned into a finite set V of concrete nodes and a set $I = \{1, 2, \dots, n\}$ of abstract nodes given by $n \in \mathbb{N}$. Moreover, $\rightarrow \subseteq W \times \text{ACT} \times W$ is the labeled edge relation, E is a finite set of hyperedges, and $\alpha: E \rightarrow W^*$ assigns a sequence of attached nodes to every hyperedge. Finally, $\gamma: V \rightarrow 2^{\text{COL}}$ assigns a set of colors to every concrete node and $\ell: E \rightarrow \text{LAB}$ assigns labels to hyperedges. We denote by **HG** the set of all hypergraphs.

We write $u \xrightarrow{a} v$ instead of $(u, a, v) \in \rightarrow_{\mathcal{H}}$. Moreover, we write $V_{\mathcal{H}}, n_{\mathcal{H}}, W_{\mathcal{H}}$, and so on, to refer to the components of a hypergraph $\mathcal{H}$. Figure 2 (left) illustrates an HG with four concrete nodes $v_1 - v_4$ with colors **blue**, **red**, or **green**, two abstract nodes 1 and 2, one hyperedge e with label A , and six ordinary edges indicated by arrows. For simplicity, we omit ordinary edge labels. The numbered connections to e indicate the sequence of attached nodes, i.e. $\alpha(e) = 1v_2v_3$.

Hyperedges act as placeholders for HGs that can be inserted into the current hypergraph, while abstract nodes serve as connection points through which the current hypergraph can be embedded into other HGs. An HG with neither abstract nodes nor hyperedges is a labeled transition system (LTS) [5, p. 20].³ We

³ The described LTS has no initial states, but one can use a color to model them.

reason about ω -regular properties of such LTSs by considering their traces, i.e. sequences of node colors. To formalize traces, we first define the set $\mathbf{Paths}_{\mathcal{H}}$ of all *paths* of $\mathcal{H} \in \mathbf{HG}$ using ordinary edges:

$$\mathbf{Paths}_{\mathcal{H}} = \left\{ \bar{p} \in W_{\mathcal{H}}^* \uplus W_{\mathcal{H}}^{\omega} \mid \forall i \in [0, |\bar{p}|). \exists a \in \mathbf{ACT}. p_i \xrightarrow{a}_{\mathcal{H}} p_{i+1} \right\}$$

A trace is then obtained from a path by considering the colors of each node. Formally, the set $\mathbf{Traces}_{\mathcal{H}}^*(u, v)$ of *finite traces* in $\mathcal{H}$ from $u \in W_{\mathcal{H}}$ to $v \in W_{\mathcal{H}}$ is

$$\mathbf{Traces}_{\mathcal{H}}^*(u, v) = \{ \gamma_{\mathcal{H}}(u \bar{p} v|_{V_{\mathcal{H}}}) \mid u \bar{p} v \in \mathbf{Paths}_{\mathcal{H}} \text{ and } \bar{p} \in V_{\mathcal{H}}^* \}.$$

Notice that traces do not permit abstract nodes in the middle of a path. The set $\mathbf{Traces}_{\mathcal{H}}^{\omega}(u)$ of *infinite traces* in $\mathcal{H}$ starting in $u \in W_{\mathcal{H}}$ is

$$\mathbf{Traces}_{\mathcal{H}}^{\omega}(u) = \{ \gamma_{\mathcal{H}}(u \bar{p}|_{V_{\mathcal{H}}}) \mid u \bar{p} \in \mathbf{Paths}_{\mathcal{H}} \text{ and } \bar{p} \in V_{\mathcal{H}}^{\omega} \}.$$

Figure 2 (right) depicts examples of sets of traces for the HG on the left. The set $\mathbf{Traces}_{\mathcal{H}}^{\omega}(v_1)$ is empty, because every infinite path starting in v_1 has to pass through the abstract node 2.

As for transition systems, an HG $\mathcal{H}$ and a node $v \in W_{\mathcal{H}}$ *satisfy* a property given by a Büchi automaton $\mathcal{M} \in \mathbf{BUCHI}(2^{\mathbf{COL}})$ iff all infinite traces starting in v are accepted by $\mathcal{M}$, i.e. $\mathcal{H}, v \models \mathcal{M}$ iff $\mathbf{Traces}_{\mathcal{H}}^{\omega}(v) \subseteq \mathcal{L}(\mathcal{M})$.

2.3 Hyperedge Replacement Grammars

To describe families of transition systems, we define how hyperedges are replaced by HGs. A (hyperedge) *assignment* is a function $\bar{\mathcal{K}}: E_{\mathcal{H}} \rightarrow \mathbf{HG}$ that maps every hyperedge $e \in E_{\mathcal{H}}$ with k attached nodes to an HG with k abstract nodes, i.e. $|\alpha_{\mathcal{H}}(e)| = n_{\bar{\mathcal{K}}(e)}$. Hyperedge replacement then substitutes every hyperedge e by the graph $\bar{\mathcal{K}}(e)$ while identifying the i -th attached node of e with the i -th abstract node of $\bar{\mathcal{K}}(e)$. Formally, the HG $\mathcal{H}[\bar{\mathcal{K}}] = (V, n_{\mathcal{H}}, \rightarrow, E, \alpha, \gamma, \ell)$ obtained from *hyperedge replacement* according to assignment $\bar{\mathcal{K}}$ is given by:

$$V = V_{\mathcal{H}} \uplus \biguplus_{e \in E_{\mathcal{H}}} V_{\bar{\mathcal{K}}(e)} \quad E = \biguplus_{e \in E_{\mathcal{H}}} E_{\bar{\mathcal{K}}(e)} \quad \gamma = \gamma_{\mathcal{H}} \uplus \biguplus_{e \in E_{\mathcal{H}}} \gamma_{\bar{\mathcal{K}}(e)} \quad \ell = \biguplus_{e \in E_{\mathcal{H}}} \ell_{\bar{\mathcal{K}}(e)}$$

Moreover, for every $e \in E_{\mathcal{H}}$, let $f_e(i)$ identify the i -th abstract node of $\bar{\mathcal{K}}(e)$ with the i -th node attached to e , i.e. $f_e(i) = \alpha_{\bar{\mathcal{K}}(e)_i}$ if $i \in I_{\bar{\mathcal{K}}(e)}$ and $f_e(v) = v$ if $v \in V_{\bar{\mathcal{K}}(e)}$. Then the edge relation $\rightarrow$ and assignment of attached nodes α are

$$\rightarrow = \rightarrow_{\mathcal{H}} \uplus \biguplus_{e \in E_{\mathcal{H}}} \{ (f_e(u), a, f_e(v)) \mid u \xrightarrow{a}_{\bar{\mathcal{K}}(e)} v \} \quad \text{and} \quad \alpha = \biguplus_{e \in E_{\mathcal{H}}} f_e \circ \alpha_{\bar{\mathcal{K}}(e)}.$$

If an HG $\mathcal{H}$ has only a single hyperedge, which we replace by $\mathcal{K}$, we write $\mathcal{H}[\mathcal{K}]$ instead of $\mathcal{H}[\lambda e. \mathcal{K}]$. Throughout this paper, we consider HGs up to isomorphism. Hence, nodes and hyperedges can always be renamed before replacement.

A trivial example of hyperedge replacement is the identity. The corresponding assignment maps every hyperedge with label A and k attached nodes to the k -handle $A^\circ = (\emptyset, k, \emptyset, \{e\}, \lambda e. (1 \dots k), \lambda e. A)$, consisting of one hyperedge labeled A and k abstract nodes attached to it in ascending order.⁴

We model families of LTSs with hyperedge replacement grammars (HRGs).

Definition 3. An HRG is a tuple $\mathcal{G} = (N, Z, P)$, where N is a finite set of nonterminals equipped with an arity $\mathbf{ar}: N \rightarrow \mathbb{N}$, $Z \subseteq N$ is a set of start symbols, and $P \subseteq N \times \mathbf{HG}$ is a set of production rules such that, for all $(A, \mathcal{H}) \in P$, (i) $\mathbf{ar}(A) = n_{\mathcal{H}}$ and (ii) for all $e \in E_{\mathcal{H}}$, $\ell(e) \in N$ and $|\alpha_{\mathcal{H}}(e)| = \mathbf{ar}(\ell(e))$.

Conditions (i) and (ii) ensure for every production rule $(A, \mathcal{H}) \in P$ that replacing a hyperedge labelled with A by $\mathcal{H}$ is well-defined. We will define the graph language of an HRG in terms of its derivation trees, which we introduce first.

Definition 4. Let $\mathcal{G} = (N, Z, P)$ be an HRG and $A \in N$. The set $\mathbf{Tree}_{\mathcal{G}}(A)$ of (partial) derivation trees of A is defined inductively by the following rules:

1. $A \in \mathbf{Tree}_{\mathcal{G}}(A)$ (i.e. derivations can be stopped prematurely).
2. If $(A, \mathcal{R}) \in P$ and $\bar{T}$ is a function that assigns to every $e \in E_{\mathcal{R}}$ a derivation tree $\bar{T}(e) \in \mathbf{Tree}_{\mathcal{G}}(\ell_{\mathcal{R}}(e))$, then $(\mathcal{R}, \bar{T}) \in \mathbf{Tree}_{\mathcal{G}}(A)$.

A derivation tree is complete if it is constructed without rule (1). Furthermore, it is a tree with hole A if (1) is applied exactly once and with nonterminal A .

Every partial derivation tree describes how to apply production rules to derive an HG from a nonterminal through hyperedge replacement. Formally, the HG $\langle T \rangle$ assembled from a derivation tree $T \in \mathbf{Tree}_{\mathcal{G}}(A)$ is given by

$$\langle T \rangle = \begin{cases} A^\circ, & T = A \\ \mathcal{R}[\lambda e. \langle \bar{T}(e) \rangle], & T = (\mathcal{R}, \bar{T}). \end{cases}$$

The first case stops further replacements of a hyperedge labeled A . Hence, complete derivation trees and trees with a hole yield HGs with zero and one hyperedge, respectively. The second case replaces each hyperedge by the HG assembled from its assigned derivation tree. Our formal model checking problem will be phrased in terms of the *language* of an HRG, i.e. all hypergraphs that are assembled from complete derivation trees of a starting symbol.

Definition 5. The language of an HRG $\mathcal{G} = (N, Z, P)$ and some $A \in N$ is

$$\mathcal{L}_A(\mathcal{G}) = \{ \langle T \rangle \mid T \in \mathbf{Tree}_{\mathcal{G}}(A) \text{ complete} \}.$$

Moreover, the language of HGs generated by $\mathcal{G}$ is $\mathcal{L}(\mathcal{G}) = \bigcup_{A \in Z} \mathcal{L}_A(\mathcal{G})$.

Figure 3 illustrates the language of the HRG $\mathcal{G}$ from Figure 1a for the nonterminals A and S . Since $\mathbf{ar}(A) = 2$, every HG in $\mathcal{L}_A(\mathcal{G})$ has two abstract nodes. Every HG in $\mathcal{L}_S(\mathcal{G})$ is an LTS modeling a doubly-linked list, where the rightmost node is blue and every other node is red.

⁴ We use lambda expressions to denote functions. For example, $\lambda e. A$ is the function that maps every hyperedge e to the nonterminal A .

$$\mathcal{L}_A(\mathcal{G}) = \left\{ \begin{array}{c} \mathcal{R}_1 \\ \textcircled{1} \rightarrow \textcircled{2} \end{array}, \begin{array}{c} \mathcal{R}_2[\mathcal{R}_1] \\ \textcircled{1} \rightarrow \textcircled{1} \rightarrow \textcircled{2} \end{array}, \dots \right\} \quad \mathcal{L}_S(\mathcal{G}) = \left\{ \begin{array}{c} \mathcal{R}_3[\mathcal{R}_1] \\ \textcircled{1} \rightarrow \textcircled{1} \rightarrow \textcircled{2} \end{array}, \begin{array}{c} \mathcal{R}_3[\mathcal{R}_2[\mathcal{R}_1]] \\ \textcircled{1} \rightarrow \textcircled{1} \rightarrow \textcircled{1} \rightarrow \textcircled{2} \end{array}, \dots \right\}$$

Fig. 3: Languages of the grammar from Figure 1a.

2.4 Formal Problem Statement

We now formalize the main problem studied in this paper and discuss applications to model checking and satisfiability of infinite families of finite LTSs.

From now on, we will only work with HRGs $\mathcal{G}$ whose language $\mathcal{L}(\mathcal{G})$ consists of LTSs, which is guaranteed if $\text{ar}(S) = 0$ holds for every starting symbol S . Furthermore, we denote by $\text{del}_c(\mathcal{H})$ the HG that is identical to $\mathcal{H}$ except that the node color $c \in \text{COL}$ has been deleted from every node.⁵ Our goal is then to determine, for every LTS $\mathcal{H}$ generated by an HRG, all nodes v that can be used as initial states of $\mathcal{H}$ such that $(\mathcal{H}, v)$ satisfies a given ω -regular property given by a Büchi automaton $\mathcal{M}$. We characterize those nodes by constructing a *recolored* HRG in which precisely those nodes are additionally colored with $\mathcal{M}$.

Büchi recoloring problem for HRGs. Given an HRG $\mathcal{G}$ generating LTSs with node colors COL and a Büchi automaton $\mathcal{M} \in \text{BUCHI}(2^{\text{COL}})$, compute an HRG $\mathcal{G}_{\mathcal{M}}$ using the set of node colors $\text{COL} \uplus \{\mathcal{M}\}$ such that

- a) $\text{del}_{\mathcal{M}}(\mathcal{L}(\mathcal{G}_{\mathcal{M}})) = \{\text{del}_{\mathcal{M}}(\mathcal{H}) \mid \mathcal{H} \in \mathcal{L}(\mathcal{G}_{\mathcal{M}})\} = \mathcal{L}(\mathcal{G})$ and
- b) $\forall \mathcal{H} \in \mathcal{L}(\mathcal{G}_{\mathcal{M}}) \forall v \in V_{\mathcal{H}}. \quad \mathcal{M} \in \gamma_{\mathcal{H}}(v) \iff \mathcal{H}, v \models \mathcal{M}.$

Property (a) requires that the recolored HRG generates the same LTSs apart from adding the color $\mathcal{M}$ to some nodes and property (b) requires that the right nodes are colored with $\mathcal{M}$. If we use a fixed node color *init* to indicate initial states of interest in the LTSs generated by an HRG $\mathcal{G}$, then a solution to the above problem allows deciding whether *all*, *any*, or *infinitely many* LTSs generated by $\mathcal{G}$ satisfy an ω -regular property given by a Büchi automaton $\mathcal{M}$. For example, to reason about *all* LTSs, it suffices to (1) compute the recolored HRG $\mathcal{G}_{\mathcal{M}}$, (2) remove all unproductive rules⁶, and (3) check whether for all remaining production rules that every node colored as initial is also colored with $\mathcal{M}$. Decision procedures for the other problems work similarly, but need to find a single derivation tree or a loop among production rules in step (3) with the same property as above. Finally, recoloring the grammar rather than deciding the ω -regular property alone, allows extending our approach to CTL* properties via recursive HRG recoloring; see Section 5 for details.

⁵ i.e. $\text{del}_c(\mathcal{H})$ is identical to $\mathcal{H}$ except for the node coloring function $\lambda v. \ell_{\mathcal{H}}(v) \setminus \{c\}$.

⁶ A rule is unproductive, it does not appear in any complete derivation tree of a starting symbol. Unproductive rules can be removed in linear time (cf. [27]).

$$\mathcal{C}_A(\mathcal{G}) = \left\{ \begin{array}{c} \mathcal{R}_3[A^\circ] \\ \text{Diagram 1} \end{array}, \begin{array}{c} \mathcal{R}_3[\mathcal{R}_2[A^\circ]] \\ \text{Diagram 2} \end{array}, \begin{array}{c} \mathcal{R}_3[\mathcal{R}_2[\mathcal{R}_2[A^\circ]]] \\ \text{Diagram 3} \end{array}, \dots \right\}, \quad \mathcal{C}_S(\mathcal{G}) = \left\{ \begin{array}{c} S^\circ \\ \text{Diagram 4} \end{array} \right\}$$

Fig. 4: Contexts of the grammar from Figure 1a.

3 Compositional Reasoning about HRG Languages

This section presents a generic approach for compositional reasoning about HRGs based on congruences, which we apply in Section 4 to solve the Büchi recoloring problem. The main idea is that every complete derivation tree $T \in \text{Tree}_{\mathcal{G}}(A)$ can be decomposed into three components: (1) a tree $T' \in \text{Tree}_{\mathcal{G}}(A)$ with some hole B , (2) a production rule $(B, \mathcal{R}) \in P_{\mathcal{G}}$, and (3) a complete derivation tree for every hyperedge in $\mathcal{R}$ (cf. [21]). Our goal is to reason compositionally about the HGs $\langle T \rangle$ of such trees T by considering only component (2), i.e. individual production rules $(B, \mathcal{R})$, and limited information on the components (1) and (3). The HGs obtained from (1) are also called contexts.

Definition 6. *The set of contexts of an HRG $\mathcal{G} = (N, Z, P)$ and $B \in N$ is*

$$\mathcal{C}_B(\mathcal{G}) = \{ \langle T \rangle \mid \exists A \in Z. T \in \text{Tree}_{\mathcal{G}}(A) \text{ with hole } B \}.$$

Figure 4 illustrates the sets of contexts of the HRG $\mathcal{G}$ from Figure 1a. Note that there is only one context of S , because $\mathcal{G}$ contains no hyperedge labelled S . The above decomposition of complete derivation trees can also be phrased as a decomposition of the HGs assembled from components (1) - (3).

Lemma 1. *Let $\mathcal{G}$ be a HRG, $\mathcal{H} \in \mathcal{L}(\mathcal{G})$, and $v \in V_{\mathcal{H}}$. Then, there exists a rule $(B, \mathcal{R}) \in P_{\mathcal{G}}$, a context $\mathcal{J} \in \mathcal{C}_B(\mathcal{G})$, and an assignment $\bar{\mathcal{K}}: E_{\mathcal{H}} \rightarrow \text{HG}$ with $\bar{\mathcal{K}}(e) \in \mathcal{L}_{\ell_{\mathcal{H}}(e)}(\mathcal{G})$ for all $e \in E_{\mathcal{H}}$ such that $\mathcal{H} = \mathcal{J}[\mathcal{R}[\bar{\mathcal{K}}]]$ and $v \in V_{\mathcal{R}}$.*

To formalize what we mean by limited information about (1) and (3), we define *HGs with a view*, i.e. HGs with a distinguished set of *exposed* nodes. We assume that abstract nodes and attached nodes are always exposed, as they are affected by hyperedge replacement. We allow exposing an additional subset of the remaining unattached nodes $UA_{\mathcal{H}} = \{v \in V_{\mathcal{H}} \mid \forall e \in E_{\mathcal{H}}. v \notin \alpha_{\mathcal{H}}(e)\}$.

Definition 7. *A hypergraph with a view (HGV) is a pair $(\mathcal{H}, U)$, where $\mathcal{H} \in \text{HG}$ and $U \subseteq UA_{\mathcal{H}}$. We denote by $X_{\mathcal{V}}$ the set of exposed nodes $U \cup (W_{\mathcal{H}} \setminus UA_{\mathcal{H}})$. We call $(\mathcal{H}, \emptyset)$ the trivial HGV of $\mathcal{H}$ and $(\mathcal{H}, UA_{\mathcal{H}})$ the full HGV of $\mathcal{H}$.*

We lift hyperedge replacement to HGVs by joining views, i.e.

$$(\mathcal{H}, U)[\lambda e. (\bar{\mathcal{J}}(e), U_e)] = (\mathcal{H}[\bar{\mathcal{J}}], \biguplus_e U_e \uplus U).$$

Analogously, we lift paths and traces to HGVs. Furthermore, for a Büchi automaton $\mathcal{M}$, $\mathcal{V} = (\mathcal{H}, U)$, and $x \in X_{\mathcal{V}}$, we define $\mathcal{V}, x \models \mathcal{M}$ iff $\mathcal{H}, x \models \mathcal{M}$. We characterize the information about the context $\mathcal{J}$ and the HGs $\bar{\mathcal{K}}(e)$ needed for compositional reasoning as equivalence classes of relations between HGVs. To this end, we must relate exposed nodes and hyperedges of HGVs.

Definition 8. Let $\mathcal{V} = (\mathcal{H}, U) \in \text{HGV}$ and $\mathcal{V}' = (\mathcal{H}', U') \in \text{HGV}$. A coupling between $\mathcal{V}$ and $\mathcal{V}'$ is a pair (η, μ) where $\eta : U \xrightarrow{\text{bij}} U'$ and $\mu : E_{\mathcal{H}} \xrightarrow{\text{bij}} E_{\mathcal{H}'}$ such that there exists a bijection ν for which

$$\forall i \in I_{\mathcal{H}}. \nu(i) = i, \quad \forall v \in X. \nu(v) = \eta(v), \quad \forall e \in E_{\mathcal{H}} \forall i. \nu(\alpha_{\mathcal{H}}(e)_i) = \alpha_{\mathcal{H}'}(\mu(e))_i.$$

The mappings η and μ determine how unattached nodes and hyperedges are related. The relation between abstract nodes in the two HGs is implicitly given by their numbering. The same holds for nodes attached to hyperedges. We extend η to a mapping ν that makes those correspondences explicit.

Definition 9. An HGV-relation $\equiv$ is a set of tuples $(\mathcal{U}, \mathcal{V}, \eta, \mu)$, where $\mathcal{U}, \mathcal{V}$ are HGVs and (η, μ) is a coupling. We write $\eta, \mu \vdash \mathcal{U} \equiv \mathcal{V}$ instead of $(\mathcal{U}, \mathcal{V}, \eta, \mu) \in \equiv$.

For example, isomorphisms are HGV-relations. Specifically, two views $(\mathcal{H}, U)$ and $(\mathcal{H}', U')$ are isomorphic under η and μ , written $\eta, \mu \vdash (\mathcal{H}, U) \cong (\mathcal{H}', U')$, iff $\rightarrow_{\mathcal{H}'} = \nu(\rightarrow_{\mathcal{H}})$, $\gamma_{\mathcal{H}} = \gamma_{\mathcal{H}'} \circ \eta$ and $\ell_{\mathcal{H}} = \ell_{\mathcal{H}'} \circ \mu$. The hypergraphs $\mathcal{H}$ and $\mathcal{H}'$ are isomorphic if their full HGVs are isomorphic.

The precise choice of η and μ in HGV-relations $\equiv$ is often irrelevant. Hence, we define $\mathcal{U} \equiv \mathcal{V}$ iff $\exists \eta, \mu. \eta, \mu \vdash \mathcal{U} \equiv \mathcal{V}$ that uses the same symbol $\equiv$ by slight abuse of notation. We will only work with HGV-relations $\equiv$ that are equivalence relations and respect hyperedge replacement, i.e. replacing hyperedges by equivalent HGVs in equivalent HGVs must yield equivalent HGVs again.

Definition 10. An HGV-congruence is an HGV-relation $\equiv$ such that, for all $\mathcal{U}, \mathcal{U}' \in \text{HGV}$ and hyperedge assignments $\bar{\mathcal{V}}$ on $\mathcal{U}$ and $\bar{\mathcal{V}}'$ on $\mathcal{U}'$,

$$\begin{aligned} (\eta, \mu) \vdash \mathcal{U} \equiv \mathcal{U}' \quad \text{and} \quad \forall e \in E_{\mathcal{H}}. (\eta_e, \mu_e) \vdash \bar{\mathcal{V}}(\mu(e)) \equiv \bar{\mathcal{V}}'(\mu(e)) \\ \text{implies} \quad (\biguplus_e \eta_e \uplus \eta, \biguplus_e \mu_e) \vdash \mathcal{U}[\bar{\mathcal{V}}] \equiv \mathcal{U}'[\bar{\mathcal{V}}']. \end{aligned}$$

For instance, every partial isomorphism is an HGV-congruence. If $\equiv$ is an HGV-congruence, we denote by $[\mathcal{V}]_{\equiv}$ the equivalence class of $\mathcal{V}$. For every HRG $\mathcal{G}$ and every HGV-congruence $\equiv$, one can then construct a refined HRG $\mathcal{G}_{\equiv}$. For every rule $(A, \mathcal{R})$ that appears in $\mathcal{H} = \mathcal{J}[\mathcal{R}[\bar{\mathcal{K}}]] \in \mathcal{L}(\mathcal{G})$, $\mathcal{G}_{\equiv}$ annotates A with the equivalence class $[(\mathcal{J}, \emptyset)]$, and each $\ell_{\mathcal{R}}(e)$ with $[(\bar{\mathcal{K}}(e), \emptyset)]$. These equivalence classes allow computing $[(\mathcal{J}[\mathcal{R}[\bar{\mathcal{K}}]], V_{\mathcal{R}})]$ without knowing $\mathcal{J}$ or $\mathcal{K}$. We can thus reason about $\mathcal{R}$ in the context in which it is applied.

To formalize refined HRGs, we rely on two conventions. First, we denote by $A \Rightarrow \mathcal{R}(\bar{B})$ the production rule $(A, \mathcal{R}')$, where $\mathcal{R}'$ is identical to $\mathcal{R}$ except that the hyperedge labeling function is $\bar{B}$ instead of $\ell_{\mathcal{R}}$. Second, given two functions f, g with the same domain, we denote by $f \times g$ the function that assigns the value of both functions to each domain element, i.e. $f \times g = \lambda e. (f(e), g(e))$.

Definition 11. Let $\mathcal{G} = (N, P, Z)$ be an HRG and $\equiv$ be a HGV-congruence. Then the refined HRG $\mathcal{G}_{\equiv} = (N_{\equiv}, P_{\equiv}, Z_{\equiv})$ is defined as follows:

$$\begin{aligned} N_{\equiv} &= \{(A, \mathfrak{S}, \mathfrak{C}) \mid A \in N, \mathfrak{S} \in [(\mathcal{L}_A(\mathcal{G}), \emptyset)], \mathfrak{C} \in [(\mathcal{C}_A(\mathcal{G}), \emptyset)]\} \\ P_{\equiv} &= \left\{ (A, \mathfrak{S}, \mathfrak{C}) \Rightarrow \mathcal{R}(\bar{B} \times \bar{\mathfrak{T}} \times \bar{\mathfrak{D}}) \mid \begin{array}{l} A \Rightarrow \mathcal{R}(\bar{B}) \in P, \mathfrak{S} = (\mathcal{R}, \emptyset)[\bar{\mathfrak{T}}], \\ \bar{\mathfrak{D}} = \lambda e. \mathfrak{C}[(\mathcal{R}, \emptyset)[\bar{\mathfrak{T}}\langle e/\bar{B}(e)^{\circ} \rangle]] \end{array} \right\} \\ Z_{\equiv} &= \{(A, \mathfrak{S}, [(\mathcal{A}^{\circ}, \emptyset)]) \in N_{\equiv} \mid A \in Z\}, \end{aligned}$$

where $\overline{\mathfrak{T}}\langle e/B(e)^\circ \rangle$ is the hyperedge assignment that is identical to $\overline{\mathfrak{T}}$ except that it assigns the handle $B(e)^\circ$ to e .

Note that the information about contexts and languages of hyperedges added in the above construction is *local* to each rule, i.e. it depends solely on the equivalence classes assigned to the nonterminal of the considered rule and the nonterminals assigned to hyperedges in that rule.⁷ The following theorem justifies that enriching HRGs with such information is correct for HGV-congruences.

Theorem 1. *Let $\mathcal{G}$ be an HRG and $\equiv$ be a HGV-congruence. Then:*

1. *For all $\mathcal{H} \in \mathcal{L}_{(A, \mathfrak{S}, \mathfrak{C})}(\mathcal{G}_\equiv)$, we have $\mathfrak{S} = \llbracket (\mathcal{H}, \emptyset) \rrbracket$.*
2. *For all $\mathcal{J} \in \mathcal{C}_{(A, \mathfrak{S}, \mathfrak{C})}(\mathcal{G}_\equiv)$, we have $\mathfrak{C} = \llbracket (\mathcal{J}, \emptyset) \rrbracket$.*

In other words, the refined HRG can only generate HGs and contexts from a nonterminal $(A, \mathfrak{S}, \mathfrak{C})$ that are in the equivalence classes $\mathfrak{S}$ and $\mathfrak{C}$, respectively.

4 $\mathcal{M}$ -equivalence: A Congruence for Büchi Recoloring

To solve the Büchi recoloring problem (see Section 2.4), we now define an HGV-congruence for ω -regular properties. Throughout this section, we fix a Büchi automaton $\mathcal{M} = (Q, \Sigma, \Delta, Q_0, F)$ with $\Sigma = 2^{\text{COL}}$. Our goal is to define a congruence relation on HGVs $\mathcal{V}$ that suffices to decide whether $\mathcal{M}$ accepts on the traces that start and end on exposed nodes $X_{\mathcal{V}}$. We first define such an equivalence on strings, which resembles the indistinguishability relation used in the proof of the well-known Myhill-Nerode theorem for (cf. [27,40]).

Definition 12 ($\mathcal{M}$ -equivalence for Strings). *Two strings $\overline{\sigma}, \overline{\sigma}' \in \Sigma^*$ are $\mathcal{M}$ -equivalent if $\forall p, q, b. (p, \overline{\sigma}, q, b) \in \Delta^* \iff (p, \overline{\sigma}', q, b) \in \Delta^*$. Two infinite strings $\overline{\sigma}, \overline{\sigma}' \in \Sigma^\omega$ are $\mathcal{M}$ -equivalent if $\forall p. (p, \overline{\sigma}, \text{T}) \in \Delta^\omega \iff (p, \overline{\sigma}', \text{T}) \in \Delta^\omega$. We write $\overline{\sigma} \equiv_{\mathcal{M}} \overline{\sigma}'$ to denote that $\overline{\sigma}$ and $\overline{\sigma}'$ are $\mathcal{M}$ -equivalent.*

$\mathcal{M}$ -equivalent strings cannot be distinguished by $\mathcal{M}$. We lift this notion to sets of strings. Specifically, two sets of finite (resp. infinite) strings $S, S' \subseteq \Sigma^*$ (resp. $S, S' \subseteq \Sigma^\omega$) are equivalent, iff for each string $\overline{\sigma} \in S$, there exists an $\mathcal{M}$ -equivalent string $\overline{\sigma}' \in S'$, and vice versa. To lift $\mathcal{M}$ -equivalence to HGVs $\mathcal{V}$, we consider the traces between all exposed nodes $x, y \in X_{\mathcal{V}}$. For every equivalent $\mathcal{V}'$, we then require that the sets of traces between coupled exposed nodes are equivalent.

Definition 13 ($\mathcal{M}$ -equivalence for views). *Let $\mathcal{V}, \mathcal{V}' \in \text{HGV}$ with coupling (η, μ) and mapping ν . Then $\mathcal{V}$ is $\mathcal{M}$ -equivalent to $\mathcal{V}'$ if all $u, v \in X_{\mathcal{V}}$ satisfy*

$$\text{Traces}_{\mathcal{V}}^*(x, y) \equiv_{\mathcal{M}} \text{Traces}_{\mathcal{V}'}^*(\nu(x), \nu(y)) \text{ and } \text{Traces}_{\mathcal{V}}^\omega(x) \equiv_{\mathcal{M}} \text{Traces}_{\mathcal{V}'}^\omega(\nu(x)).$$

⁷ We will illustrate refined HRGs at the end of the next section, which introduces a concrete HGV-congruence for reasoning about ω -regular properties.

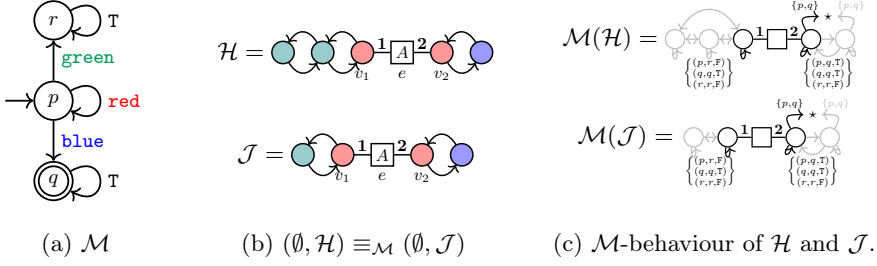


Fig. 5: Büchi automaton $\mathcal{M}$ for $(\text{red} \ U \ \text{blue})$ and its behaviour on HGs.

Figure 5a shows a Büchi automaton and Figure 5b shows two example HGs. The HGVs $(\mathcal{H}, \emptyset)$ and $(\mathcal{J}, \emptyset)$ are $\mathcal{M}$ -equivalent. To see this, note that the two graphs have the same traces that start or end in v_2 . While the respective sets $\text{Traces}^*(v_1, v_1)$ and $\text{Traces}^\omega(v_1)$ differ for $\mathcal{H}$ and $\mathcal{J}$, they are indistinguishable by $\mathcal{M}$ as all such paths transition $\mathcal{M}$ to the sink state r .

$\mathcal{M}$ -equivalence enjoys two important properties. First, it is a congruence.

Theorem 2. *$\mathcal{M}$ -equivalence is an HGV-congruence.*

Second, for $\mathcal{M}$ -equivalent graphs, the Büchi automaton $\mathcal{M}$ cannot distinguish between the traces of exposed nodes that are coupled (via ν):

Lemma 2. *Let $\mathcal{V}$ and $\mathcal{V}'$ be $\mathcal{M}$ -equivalent via the mapping ν . Then, for all exposed nodes $x \in X_{\mathcal{V}}$, we have $(\mathcal{V}, x) \models \mathcal{M} \iff (\mathcal{V}', \nu(x)) \models \mathcal{M}$.*

The above lemma justifies lifting $\models$ from HVGs to their $\mathcal{M}$ -equivalence classes.

We are now in a position to define a solution for the Büchi recoloring problem for a given HRG $\mathcal{G}$. Morally, we consider every HG $\mathcal{H} \in \mathcal{L}(\mathcal{G})$ and any node $v \in V_{\mathcal{H}}$. To check whether v needs to be colored with $\mathcal{M}$, we have to decide whether $\mathcal{H}, v \models \mathcal{M}$ holds. To this end, we decompose $\mathcal{H}$ into $\mathcal{J}[\mathcal{R}[\bar{\mathcal{K}}]]$ such that $v \in V_{\mathcal{R}}$. We can then extract the $\mathcal{M}$ -equivalence class of $\mathcal{J}$ and each $\bar{\mathcal{K}}(e)$ from the refined grammar $\mathcal{G}_{\equiv \mathcal{M}}$ in order to compute the $\mathcal{M}$ -equivalence class $[(\mathcal{J}[\mathcal{R}[\bar{\mathcal{K}}]], V_{\mathcal{R}})]$. Lemma 2, then guarantees that we can check whether the automaton $\mathcal{M}$ is satisfied on each exposed node $V_{\mathcal{R}}$, including v .

Definition 14 (Grammar recoloring). *Let $\mathcal{G}$ be an HRG. Then, we define its $\mathcal{M}$ -recoloring $\mathcal{G}_{\mathcal{M}} = (N_{\mathcal{G}_{\equiv \mathcal{M}}}, P, Z_{\mathcal{G}_{\equiv \mathcal{M}}})$ where*

$$P = \left\{ (A, \mathfrak{S}, \mathfrak{C}) \Rightarrow \mathcal{R}'(\bar{B} \times \bar{\mathfrak{T}} \times \bar{\mathfrak{D}}) \mid \begin{array}{l} (A, \mathfrak{S}, \mathfrak{C}) \Rightarrow \mathcal{R}(\bar{B} \times \bar{\mathfrak{T}} \times \bar{\mathfrak{D}}) \in P_{\mathcal{G}_{\equiv \mathcal{M}}} \wedge \mathcal{R} = \text{del}_{\mathcal{M}}(\mathcal{R}') \wedge \\ \forall v \in V_{\mathcal{R}'} . \mathcal{M} \in \gamma_{\mathcal{R}'}(v) \iff (\mathfrak{C}[(\mathcal{R}', U_{\mathcal{A}_{\mathcal{R}'}})[\bar{\mathfrak{T}}]], v) \models \mathcal{M} \end{array} \right\}$$

Theorem 3 (Soundness and Completeness). *Let $\mathcal{G}$ be a HRG. Then,*

$$\forall \mathcal{H} \in \mathcal{L}(\mathcal{G}_{\mathcal{M}}) \forall v \in V_{\mathcal{H}} . \mathcal{M} \in \gamma_{\mathcal{H}}(v) \iff (\mathcal{H}, v) \models \mathcal{M}.$$

Proof. Let $\mathcal{H} \in \mathcal{L}(\mathcal{G}_{\mathcal{M}})$, and let $v \in V_{\mathcal{H}}$. By Lemma 1, there exists a rule $(A, \mathfrak{S}, \mathfrak{C}) \Rightarrow \mathcal{R}(\overline{B} \times \overline{\mathfrak{T}} \times \overline{\mathfrak{D}}) \in P_{\mathcal{G}_{\mathcal{M}}}$, a context $\mathcal{J} \in \mathcal{C}_{(A, \mathfrak{S}, \mathfrak{C})}(\mathcal{G}_{\mathcal{M}})$ and an assignment $\overline{\mathcal{K}} : E_{\mathcal{R}} \rightarrow \mathbf{HG}$ with each $\overline{\mathcal{K}}(e) \in \mathcal{L}_{(\overline{B}(e), \overline{\mathfrak{T}}(e), \overline{\mathfrak{D}}(e))}(\mathcal{G}_{\mathcal{M}})$ such that $\mathcal{H} = \mathcal{J}[\mathcal{R}[\overline{\mathcal{K}}]]$ and $v \in V_{\mathcal{R}}$. Thus,

$$\begin{aligned} \mathcal{M} \in \gamma_{\mathcal{H}}(v) &\iff \mathcal{M} \in \gamma_{\mathcal{J}[\mathcal{R}[\overline{\mathcal{K}}]]}(v) && \text{(Lemma 1)} \\ &\iff \mathcal{M} \in \gamma_{\mathcal{R}}(v) && (v \in V_{\mathcal{R}}) \\ &\iff (\mathfrak{C}[(\mathcal{R}, UA_{\mathcal{R}})[\overline{\mathfrak{S}}]], v) \models \mathcal{M}. && \text{(Definition 14)} \end{aligned}$$

By Theorem 1, we have that $\mathfrak{C}$ is the equivalence class of $(\mathcal{J}, \emptyset)$ and $\overline{\mathfrak{S}}(e)$ is the equivalence class of $(\overline{\mathcal{K}}(e), \emptyset)$. Then, by Theorem 2, $\mathfrak{C}[[\mathcal{R}, UA_{\mathcal{R}}][\overline{\mathfrak{S}}]]$ is the equivalence class of $(\mathcal{J}[\mathcal{R}[\overline{\mathcal{K}}]], V_{\mathcal{R}})$. Hence,

$$\begin{aligned} &\iff (\mathcal{J}[\mathcal{R}[\overline{\mathcal{K}}]], v) \models \mathcal{M} && \text{(Lemma 2)} \\ &\iff (\mathcal{H}, v) \models \mathcal{M}. && \text{(Lemma 1, backwards)} \end{aligned}$$

Representing equivalence classes. In order to effectively check $\mathcal{M}$ -equivalence, we introduce a canonical representation of HGs, called $\mathcal{M}$ -behaviours.

Definition 15. *The $\mathcal{M}$ -behaviour of a hypergraph $\mathcal{H}$ is the HG*

$$\mathcal{M}(\mathcal{H}) = (V_{\mathcal{H}} \uplus \{\star\}, n_{\mathcal{H}}, \rightsquigarrow^* \uplus \rightsquigarrow^{\omega}, E_{\mathcal{H}}, \alpha_{\mathcal{H}}, \perp, \perp),$$

where $\rightsquigarrow^* \subseteq W \times 2^{Q \times Q \times \mathbf{BOOL}} \times W$ and $\rightsquigarrow^{\omega} \subseteq W \times 2^Q \times \{\star\}$ are given by

$$\begin{aligned} u \rightsquigarrow_f^* v &\iff \exists \overline{c} \in \mathbf{Traces}_{\mathcal{H}}^*(u, v). f = \{(p, q, b) \mid \Delta^*(p, \overline{c}, q, b)\} \\ u \rightsquigarrow_f^{\omega} \star &\iff \exists \overline{c} \in \mathbf{Traces}_{\mathcal{H}}^{\omega}(u). f = \{p \mid \Delta^{\omega}(p, \overline{c}, \mathbf{T})\}. \end{aligned}$$

Figure 5c depicts the $\mathcal{M}$ -behaviours of each HG in Figure 5b. Intuitively, an $\mathcal{M}$ -behavior of $\mathcal{H}$ captures, for every pair of nodes u, v whether $\mathcal{M}$ can transition from state p to q when reading any trace on a path from u to v . Moreover, for each loop in $\mathcal{H}$, it captures whether $\mathcal{M}$ accepts the corresponding infinite trace.⁸

The following theorem shows that checking $\mathcal{M}$ -equivalence boils down to checking isomorphisms on $\mathcal{M}$ -behaviours with views.

Theorem 4. *For all $(\mathcal{H}, U), (\mathcal{H}', U') \in \mathbf{HGV}$, we have*

$$\begin{aligned} &(\eta, \mu) \vdash (\mathcal{H}, U) \equiv_{\mathcal{M}} (\mathcal{H}', U') \\ \text{iff } &(\eta \uplus \{(\star, \star)\}, \mu) \vdash (\mathcal{M}(\mathcal{H}), U \uplus \{\star\}) \cong (\mathcal{M}(\mathcal{H}'), U' \uplus \{\star\}). \end{aligned}$$

For our example in Figure 5, the black portion of the $\mathcal{M}$ -behaviours in Figure 5c is exposed, showing that $(\mathcal{H}, \emptyset)$ and $(\mathcal{J}, \emptyset)$ are indeed $\mathcal{M}$ -equivalent.

We use the above result to reason about the number of equivalence classes that can be constructed during grammar recoloring.

⁸ A procedure to construct $\mathcal{M}$ -behaviours is provided in [35, A.1].

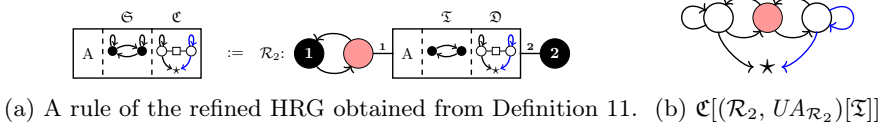


Fig. 6: Refining and recoloring an HRG rule from Figure 1a with respect to F^b .

Theorem 5. *For every HRG $\mathcal{G}$, we have*

$$|[(\mathcal{L}_A(\mathcal{G}), \emptyset)]_{\equiv \mathcal{M}}| + |[(\mathcal{C}_A(\mathcal{G}), \emptyset)]_{\equiv \mathcal{M}}| = \mathcal{O}(2^{(\text{ar}(A)^2 \times 2^{|\mathcal{Q}|^2})}).$$

Furthermore, for the recolored HRG $\mathcal{G}_{\mathcal{M}}$ with maximum number of hyperedges in rules $|E|$ and maximum arity of nonterminals $|I|$, we have

$$|\mathcal{G}_{\mathcal{M}}| = \mathcal{O}(|P| \times |N|^{|E|} \times 2^{(|E| \times |I|^2 \times 2^{|\mathcal{Q}|^2})}).$$

To conclude this section, we illustrate how refinement and recoloring are applied to the HRG in Figure 1a and the $\mathcal{M}$ -equivalence obtained from F^b . Figure 6a shows one out of 11 rules of the refined HRG; the other rules are found in [35, A.2]. The rule's nonterminals have been augmented by two $\mathcal{M}$ -equivalence classes, representing (1) the behaviour of all graphs in the language generated by the nonterminal, and (2) the behaviour of all contexts generated by the nonterminal. Similarly to Figure 5c, we illustrate the portion of $\mathcal{M}$ -behaviours exposed by the empty view. For simplicity, we represent paths that (do not) pass through a blue node by blue (black) arrows. The rule in Figure 6a satisfies the constraints of Definition 11, because $\mathcal{S} = (\mathcal{R}_2, \emptyset)[\mathcal{T}]$ and $\mathcal{D} = \mathcal{C}[(\mathcal{R}_2, \emptyset)]$. After computing the refined HRG, we apply Definition 14 by computing $\mathcal{C}[(\mathcal{R}, UA_{\mathcal{R}})[\mathcal{T}]]$ for each rule. Figure 6b illustrates the result for the rule in Figure 6a. Since there exists a trace that starts in the node added by $\mathcal{R}_2$ (red) and takes only black edges, the node does not satisfy F^b . Hence, we do not add a new color to it.

5 The recoloring algorithm for LTL and CTL*

We now present algorithms for recoloring HRGs with respect to formulae in LTL and CTL* (cf. [5]) based on $\mathcal{M}$ -behaviours. These algorithms immediately yield decision procedures for LTL and CTL* model checking of families. Figure 7 summarizes our algorithms. We denote by $\text{labels}(P)$ an auxiliary function that returns all nonterminals that appear in the set of production rules P .

The algorithm **Recolor_{LTL}** takes as input an HRG $\mathcal{G}$ and an LTL formula φ . It then constructs the corresponding recolored HRG as described in Section 4. The algorithm first converts φ into a Büchi automaton $\mathcal{M}$. It then invokes the procedure **Annotate₁**, which annotates each nonterminal of the grammar with the first of the two equivalence classes, i.e. equivalence classes representing the graphs plugged into the hyperedges of production rules. To this end, **Annotate₁**

<pre> Fun Annotate₁ ($\mathcal{G} : \text{HRG}, \mathcal{M} : \text{BUCHI}$) : HRG $P \leftarrow \text{compute closure of}$ $A \Rightarrow \mathcal{R}(\overline{B}) \in P_{\mathcal{G}}$ $\frac{(\overline{B}(e), \overline{\mathfrak{T}}(e)) \in \text{labels}(P) \quad \mathfrak{S} = (\mathcal{R}, \emptyset)[\overline{\mathfrak{T}}]}{A \times \mathfrak{S} \Rightarrow \mathcal{R}(\overline{B} \times \overline{\mathfrak{T}}) \in P}$ $Z \leftarrow \{(A, \mathfrak{S}) \mid A \in Z_{\mathcal{G}}\};$ return ($\text{labels}(P), P, Z$); </pre>	<pre> Fun Recolor_{LTL} ($\mathcal{G}_0 : \text{HRG}, \Psi : \text{LTL}$) : HRG $\mathcal{M} \leftarrow \text{Büchi}(\Psi);$ $\mathcal{G}_1 \leftarrow \text{Annotate}_1(\mathcal{G}_0, \mathcal{M});$ $\mathcal{G}_2 \leftarrow \text{Annotate}_2(\mathcal{G}_1, \mathcal{M});$ $\text{add}_{\Psi}(\mathcal{G}_2, \lambda((A, \mathfrak{S}, \mathfrak{C}) \Rightarrow \mathcal{R}(\overline{B} \times \overline{\mathfrak{T}} \times \overline{\mathfrak{D}}), v). \mathfrak{C}[\mathcal{R}[\overline{\mathfrak{T}}]], v \models \mathcal{M})$ </pre>
<pre> Fun Annotate₂ ($\mathcal{G} : \text{HRG}, \mathcal{M} : \text{BUCHI}$) : HRG $N_0 \leftarrow \{(A, \mathfrak{S}, \llbracket \emptyset, A^\circ \rrbracket) \mid (A, \mathfrak{S}) \in Z_{\mathcal{G}}\};$ $P \leftarrow \text{compute closure of}$ $(A, \mathfrak{S}) \Rightarrow \mathcal{R}(\overline{B} \times \overline{\mathfrak{T}}) \in P_{\mathcal{G}}$ $(A, \mathfrak{S}, \mathfrak{C}) \in \text{labels}(P) \cup N_0$ $\frac{\overline{\mathfrak{D}} = \lambda e. \mathfrak{C}[(\mathcal{R}, \emptyset)[\overline{\mathfrak{T}}(e/B(e)^\circ)]}{(A, \mathfrak{S}, \mathfrak{C}) \Rightarrow \mathcal{R}(\overline{B} \times \overline{\mathfrak{T}} \times \overline{\mathfrak{D}}) \in P}$ $Z \leftarrow \{(A, \mathfrak{S}, \mathfrak{C}) \mid (A, \mathfrak{S}) \in Z_{\mathcal{G}}\};$ return ($\text{labels}(P) \cup N_0, P, Z$); </pre>	<pre> Fun Recolor_{CTL*} ($\mathcal{G}_0 : \text{HRG}, \Psi : \text{CTL}^*$) : HRG switch Φ do case $\neg\Phi_1$ do $\mathcal{G}_1 \leftarrow \text{Recolor}_{\text{CTL}^*}(\mathcal{G}_0, \Phi_1);$ $\text{add}_{\neg\Phi_1}(\mathcal{G}_1, \lambda((\_, \mathcal{R}), v). \Phi_1 \notin \gamma_{\mathcal{R}}(v))$ case $\Phi_1 \wedge \Phi_2$ do $\mathcal{G}_1 \leftarrow \text{Recolor}_{\text{CTL}^*}(\mathcal{G}_0, \Phi_1);$ $\mathcal{G}_2 \leftarrow \text{Recolor}_{\text{CTL}^*}(\mathcal{G}_1, \Phi_2);$ $\text{add}_{\Phi_1 \wedge \Phi_2}(\mathcal{G}_2, \lambda((\_, \mathcal{R}), v). \{\Phi_1, \Phi_2\} \subseteq \gamma_{\mathcal{R}}(v))$ case $\forall\Psi[\Phi_1, \dots, \Phi_n] \in \text{LTL}$ do for $i \in [1..n]$ do $\mathcal{G}_i \leftarrow \text{Recolor}_{\text{CTL}^*}(\mathcal{G}_{i-1}, \Phi_i);$ end $\text{Minimize}(\text{Recolor}_{\text{LTL}}(\mathcal{G}_n, \Psi))$ end </pre>

Fig. 7: Algorithms for HRG recoloring wrt. LTL and CTL* formulae.

iteratively computes production rules enriched by equivalence classes by applying HRG rules to those obtained in previous iterations (starting with an empty set of rules). Intuitively, this process corresponds to a bottom-up computation of the equivalence classes of progressively larger elements of the HRG's language.

After **Annotate**₁ has reached a fixed point, algorithm **Recolor**_{LTL} calls the procedure **Annotate**₂ which assigns to each nonterminal its second equivalence class, i.e. the context in which rules are applied. As before, new equivalence classes are derived by repeatedly applying grammar rules to the previously obtained ones, until a fixed point is reached. Dually to **Annotate**₁, this process intuitively corresponds to a top-down computation of the equivalence classes of progressively larger *contexts*, using the equivalence classes computed by **Annotate**₁ to fill all but one of the holes. Since the $\mathcal{M}$ -equivalence relation only allows for finitely many equivalence classes, the procedure is guaranteed to terminate, evaluating to the refined grammar $\mathcal{G}_{\mathcal{M}}$ from Definition 11.

Finally, **Recolor**_{LTL} applies the function add_{Ψ} to color all relevant nodes with φ . Specifically, add_{Ψ} takes as input a function f that maps rule-node pairs $(\mathcal{R}, v)$ to Boolean values, and colors v with φ iff $f(\mathcal{R}, v)$ evaluates to true. The considered condition is the same as in Definition 14, see Figure 7.

The algorithm **Recolor**_{CTL*} in Figure 7 (bottom right) shows how to extend our LTL recoloring algorithm to CTL*, inspired by a standard algorithm for turning LTL model checkers into CTL* model checkers [5, p. 427]. Given an HRG and a CTL* formula Ψ , **Recolor**_{CTL*} reasons about CTL* properties recursively. Whenever we need to recolor a formula of the form $\forall\Psi$ containing CTL* subformulae $\Phi_1, \dots, \Phi_n$, we first recolor the HRG according to each property Φ_i , and then check $\forall\Psi$ as if each Φ_i were an atomic property (i.e. by reasoning directly on the colors). If the CTL* formula under consideration is instead a conjunction or a negation of CTL* subformulae, the grammar is first recolor

according to those subformulae, and then each rule is recolored locally (e.g. a node is colored $\neg\alpha$ iff it is not colored α). To reduce state space explosion, we minimize⁹ the HRG after every invocation of `RecolorLTL`, since the annotations on nonterminals are no longer needed at that point.

As discussed in Section 2.4, once we have computed a recolored HRG, model checking LTL or CTL* on families is straightforward. The same holds for the qualitative fragment of probabilistic computation tree logic (qPCTL) because, for finite-state models, qPCTL can be embedded into CTL [5, p. 791].

6 Implementation and Evaluation

We implemented a prototypical tool in Haskell (ca. 2k LOC) that covers our recoloring algorithms from Section 5 and LTL, CTL*, and qPCTL model checkers for families. To translate LTL formulae into Büchi automata, we use LTL2BA [25]. Furthermore, our tool applies the following optimizations: (i) To minimize HRGs after recoloring, we first translate them into tree automata accepting the grammar’s derivation trees and then reduce their size using the VATA library [32,1] with the MinOTAut module. We also implemented a more expensive standard minimization algorithm for tree automata [13, p. 37]. (ii) We prune the recolored HRG whenever colors are known to be irrelevant for subsequent recolorings. (iii) The LTL recoloring algorithm is applied to subformulae in parallel and merges the results afterward. This optimization both enables concurrent execution and yields smaller grammars, because fewer colors are required.

We evaluated our tool on 56 model checking benchmarks for various LTL, CTL* and qPCTL formulae and four families inspired by the literature. Figure 8 shows an excerpt of our experimental evaluation; the full table is found in [35, A.3]. All experiments were performed on a MacBook Pro M4 Pro with 24 GB RAM. We run experiments on the following families:

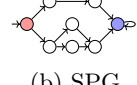
- Labeled transition systems representing Markov chain models of the *IPv4 zeroconf* protocol (cf. [30] and [5, p. 752]) for any number $k \in \mathbb{N}$ of probes and some fixed probability $p > 0$.
- *Trees of arbitrary arity* and with arbitrary colors, e.g. Figure 8a, where properties are checked from the root node. This benchmark effectively turns our tool into a CTL* satisfiability checker for tree models, since extracting a witness from the recolored grammar only requires finding a derivation tree which uses a rule that has been appropriately recolored.
- *Series parallel graphs* [16,2] of arbitrary size. See Figure 8b.
- *Sierpinski triangles* [36] of arbitrary size, with the top-most, left-most and right-most nodes identified by a different color, see Figure 8c for an example.
- *Robbery* is a variant of the SafeLock case study from [6], which considers software product lines for security-related applications. This system models an attempt to rob a bank and open a safe. Multiple features represent possibilities to do so, e.g. guessing the safe’s combination, blackmail, gathering

⁹ This is an optional step. We discuss implementation details in Section 6.

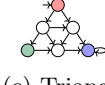
Case	Time (s)	Sat	Fal	R. #N	R. #P	M. #N	M. #P	#E	#I
<i>IPu4 zeroconf</i>									
$(\forall FG \mathbf{b}) \vee (\forall G (\exists F \mathbf{r}))$	0.67	∞	0	5	8	2	3	1	3
$X_{>0}(X_{>0}(X_{>0} \mathbf{b}))$	0.61	<i>fin</i>	∞	12	18	5	9		
$G_{=1}(F_{>0}(X_{=1}(\mathbf{r} \vee \mathbf{b}) \wedge X_{>0}(\neg \mathbf{b} \wedge F_{>0} \mathbf{b}))$	1.04	0	∞	5	8	2	3		
<i>Trees of arbitrary arity</i>									
$\neg \exists (\mathbf{b} \ U \ \neg \mathbf{b})$	0.59	∞	∞	17	140	5	22	2	1
$\forall G(\mathbf{r} \vee \neg \mathbf{b})$	0.42	∞	∞	9	74	5	22		
$\neg(\exists F \mathbf{b}) \wedge \forall G(\mathbf{r} \vee \neg \mathbf{b}) \wedge \forall G(\neg \mathbf{r})$	0.91	0	∞	9	74	5	22		
<i>Series parallel graphs</i>									
$\neg \exists (\mathbf{b} \ U \ \neg \mathbf{b})$	0.46	0	∞	16	98	2	4	2	2
$\forall G(\mathbf{r} \vee \neg \mathbf{b})$	0.45	0	∞	16	98	2	4		
$\neg(\exists F \mathbf{b}) \wedge \forall G(\mathbf{r} \vee \neg \mathbf{b}) \wedge \forall G(\neg \mathbf{r})$	0.83	0	∞	16	98	2	4		
<i>Sierpinski triangles</i>									
$\exists F \mathbf{b}$	0.46	∞	0	13	56	2	3	3	3
$\forall G(\exists F \mathbf{b})$	0.60	∞	0	13	56	2	3		
$\exists F(\exists G \mathbf{b})$	0.74	∞	0	13	56	2	3		
<i>Bank attack trees</i>									
$\exists F \mathbf{r}$	17.04	∞	∞	78	1225	13	41	3	7
$\forall G \neg \mathbf{g}$	9.80	∞	∞	30	421	9	26		
$\exists X^{\leq 2}(\mathbf{r} \vee \mathbf{b} \vee \mathbf{g}) \implies \exists F \mathbf{g}$	298.45	∞	<i>fin</i>	78	1225	13	41		
<i>Dining cryptographers</i>									
Correctness	42.65	∞	0	19	35	3	7	1	4



(a) Tree



(b) SPG



(c) Triangle

Fig. 8: An excerpt of our experiments. **Time** is the total runtime. **Sat** is the number of models in which the property is satisfied (0, *finitely* many, or ∞), **Fal** is the dual number of family members that do not satisfy property. **R.** **#P** and **R.** **#N** are the number of rules and nonterminals of the refined HRG. **M.** **#P** and **M.** **#N** indicate the number of rules and nonterminals after minimization of the recolored grammar; the original HRG's size is in the grammar's row. **#E** and **#I** are the maximum number of hyperedges and abstract nodes per rule.

information, or brute force. Our model considers different times for those approaches and admits failures, repetitions of attempts, and the potential of being spotted and jailed. Among the verified properties are: (1) one is never jailed; (2) there exists an approach to open the safe; (3) fast and dangerous attempts will always mean that one eventually ends up in jail.

- We prove correctness of the *Dining cryptographers* protocol [10] for an arbitrary number of participants.

Overall, our prototype manages to verify properties for all six families, often in less than a second. Unsurprisingly (see Theorem 5), performance does not scale well with the maximum number of hyperedges and external nodes.

7 Related Work

There is a vast body of literature on the verification of infinite-state systems based on, amongst others, pushdown automata (cf. [39]), tree automata [8], rewrite systems [33,19], programs [14], and hierarchical models [3,34]. This paper uses graph grammars for modeling and reasoning about *infinite families of finite-state systems*. We focus on two classes of related work: (1) model checking with graph grammars and (2) verification of families of finite-state models.

Reasoning about graph grammars. Burkart and Quemener [9] extended the state-labeling algorithm for CTL model checking to reason about infinite-state

Kripke structures. Apart from considering CTL instead of CTL*, they use HRGs to model a single infinite-state Kripke structure rather than an infinite family of finite-state systems. Consequently, their HRGs consist of a single production rule with a single nonterminal hyperedge, which severely restricts expressiveness.

In principle, one can obtain a decision procedure for CTL* model checking of HRG families by encoding CTL* in monadic second order logic (MSO, cf. [31]) and then applying Courcelle’s theorem [15], which states that it is decidable (with non-elementary complexity) whether all graphs generated by an HRG satisfy a given MSO formula over graphs. By contrast, we developed and implemented a direct decision procedure based on the novel abstraction of $\mathcal{M}$ -behaviours. Our procedure yields a recolored HRG with the same production rules: The colors of every node in every rule are extended according to the satisfied CTL* formulae. Hence, our algorithm naturally extends the classical state-labeling algorithm for CTL* model checking from finite-state systems to HRG families.

Groove [37,38] supports CTL model checking of graph languages by explicit enumeration. The implemented algorithms are not decision procedures.

Families of finite-state models appear in various application areas. Software product lines (SPLs, cf. [41]) are families of models obtained by composing product features that may be switched on or off. Ensuring that all possible product configurations are safe thus amounts to reasoning about all family members. A common approach is to assume that a system with all features is finite-state and then attempt to check all subsystems [12,11]. However, SPLs can describe infinite families, e.g. if features can be added to a system multiple times. Pragmatic approaches thus fall back to testing a sample of all possible subsystems [28]. This paper enables compositional CTL* model checking of infinite SPLs represented by graph grammars, which are used as a modeling formalism for SPLs [17].

For distributed systems, family verification is better known as parametric verification and has been extensively studied (cf. [22,7]). While parametric verification has not been the focus of this work, some distributed systems have been modeled by HRGs [23] and can thus, in principle, be verified with our algorithm.

Finally, our algorithm enables both checking whether all or *some* family member satisfies a CTL* formula. Hence, it can be used to check CTL* satisfiability with respect to models of interest: Given a formula and an HRG family, e.g. only tree-shaped or circular systems, does there exist a model of the formula that belongs to the family? Such questions arise, for example, in bounded synthesis, which searches for a small system that satisfies an LTL formula (cf. [18,29]).

Conclusion. We studied model checking algorithms for graph grammars modeling infinite families of finite-state labeled transition systems against temporal properties written in CTL*. Our algorithms verify whether all, some, or (in)finitely many family members satisfy a CTL* property. We implemented our approach and presented on early experiments. Future work includes improving grammar minimization and extending our approach to quantitative properties, e.g. simple fragments of PCTL, of families of Markov models.

Acknowledgements. We thank Alberto Lluch Lafuente for pointing us to the SafeLock case study.

Data-Availability Statement

Our prototype software as well as all data and instructions required to reproduce our experiments are available online:

<https://zenodo.org/records/18175332>

References

1. Almeida, R., Holík, L., Mayr, R.: Reduction of nondeterministic tree automata. In: Chechik, M., Raskin, J. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Lecture Notes in Computer Science*, vol. 9636, pp. 717–735. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_46, https://doi.org/10.1007/978-3-662-49674-9_46
2. Alur, R., Stanford, C., Watson, C.: A robust theory of series parallel graphs. *Proc. ACM Program. Lang.* **7**(POPL), 1058–1088 (2023)
3. Alur, R., Yannakakis, M.: Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.* **23**(3), 273–303 (2001)
4. Apel, S., Batory, D.S., Kästner, C., Saake, G.: *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer (2013)
5. Baier, C., Katoen, J.: *Principles of model checking*. MIT Press (2008)
6. ter Beek, M.H., Legay, A., Lluch-Lafuente, A., Vandin, A.: A framework for quantitative modeling and analysis of highly (re)configurable systems. *IEEE Trans. Software Eng.* **46**(3), 321–345 (2020)
7. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: *Decidability of Parameterized Verification. Synthesis Lectures on Distributed Computing Theory*. Morgan & Claypool Publishers (2015)
8. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular (tree) model checking. *Int. J. Softw. Tools Technol. Transf.* **14**(2), 167–191 (2012)
9. Burkart, O., Quemener, Y.: Model-checking of infinite graphs defined by graph grammars. In: *INFINITY. Electronic Notes in Theoretical Computer Science*, vol. 5, p. 16. Elsevier (1996)
10. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology* **1**, 65–75 (1988)
11. Chrszon, P., Dubslaff, C., Klüppelholz, S., Baier, C.: Profeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects Comput.* **30**(1), 45–75 (2018)
12. Classen, A., Heymans, P., Schobbens, P., Legay, A.: Symbolic model checking of software product lines. In: *ICSE*. pp. 321–330. ACM (2011)
13. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Löding, C., Tison, S., Tommasi, M.: *Tree automata techniques and applications* (2008)
14. Cook, B., Khlaaf, H., Piterman, N.: On automation of ctl^* verification for infinite-state systems. In: *CAV (1). Lecture Notes in Computer Science*, vol. 9206, pp. 13–29. Springer (2015)
15. Courcelle, B., Engelfriet, J.: *Graph structure and monadic second-order logic: a language-theoretic approach*, vol. 138. Cambridge University Press (2012)

16. Drewes, F., Kreowski, H., Habel, A.: Hyperedge replacement, graph grammars. In: Handbook of Graph Grammars, pp. 95–162. World Scientific (1997)
17. Du, X., Jiao, J., Tseng, M.M.: Graph grammar based product family modeling. *Concurr. Eng. Res. Appl.* **10**(2), 113–128 (2002)
18. Ehlers, R.: Symbolic bounded synthesis. *Formal Methods in System Design* **40**(2), 232–262 (2012)
19. Eker, S., Meseguer, J., Sridharanarayanan, A.: The maude ltl model checker. *Electronic Notes in Theoretical Computer Science* **71**, 162–187 (2004)
20. Emerson, E.A., Halpern, J.Y.: "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM* **33**(1), 151–178 (1986)
21. Engelfriet, J.: Context-free graph grammars. In: Handbook of formal languages: volume 3 beyond words, pp. 125–213. Springer (1997)
22. Esparza, J.: Keeping a crowd safe: On the complexity of parameterized verification (invited talk). In: STACS. LIPIcs, vol. 25, pp. 1–10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2014)
23. Ferrari, G., Hirsch, D., Lanese, I., Montanari, U., Tuosto, E.: Synchronised hyperedge replacement as a model for service oriented computing. In: FMCO. Lecture Notes in Computer Science, vol. 4111, pp. 22–43. Springer (2005)
24. Foerster, K., Schmid, S.: Survey of reconfigurable data center networks: Enablers, algorithms, complexity. *SIGACT News* **50**(2), 62–79 (2019)
25. Gastin, P., Oddoux, D.: Fast LTL to büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18–22, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2102, pp. 53–65. Springer (2001). https://doi.org/10.1007/3-540-44585-4_6, https://doi.org/10.1007/3-540-44585-4_6
26. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. *Formal Aspects Comput.* **6**(5), 512–535 (1994)
27. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation. *Acm Sigact News* **32**(1), 60–65 (2001)
28. Kowal, M., Schaefer, I., Tribastone, M.: Family-based performance analysis of variant-rich software systems. In: Software Engineering & Management. LNI, vol. P-239, pp. 61–62. GI (2015)
29. Kupferman, O., Lustig, Y., Vardi, M.Y., Yannakakis, M.: Temporal synthesis for bounded systems and environments. In: STACS. LIPIcs, vol. 9, pp. 615–626. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2011)
30. Kwiatkowska, M.Z., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. *Formal Methods Syst. Des.* **29**(1), 33–78 (2006)
31. Laroussinie, F., Markey, N.: Quantified CTL: expressiveness and complexity. *Log. Methods Comput. Sci.* **10**(4) (2014)
32. Lengál, O., Simáček, J., Vojnar, T.: VATA: A library for efficient manipulation of non-deterministic tree automata. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7214, pp. 79–94. Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_7, https://doi.org/10.1007/978-3-642-28756-5_7
33. Löding, C.: Model-checking infinite systems generated by ground tree rewriting. In: FoSSaCS. Lecture Notes in Computer Science, vol. 2303, pp. 280–294. Springer (2002)

34. Lohrey, M.: Model-checking hierarchical structures. *J. Comput. Syst. Sci.* **78**(2), 461–490 (2012)
35. Pettinau, R., Matheja, C.: CTL* model checking on infinite families of finite-state labeled transition systems (Technical report). arXivpreprint arXiv:2601.15756 (2026)
36. Plump, D.: The graph programming language gp. In: *International Conference on Algebraic Informatics*. pp. 99–122. Springer (2009)
37. Rensink, A.: Model checking graph grammars. In: *3rd Workshop on Automated Verification of Critical Systems, AVoCS 2003*. pp. 150–160. University of Southampton (2003)
38. Rensink, A.: Explicit state model checking for graph grammars. In: *Concurrency, Graphs and Models. Lecture Notes in Computer Science*, vol. 5065, pp. 114–132. Springer (2008)
39. Song, F., Touili, T.: Efficient CTL model-checking for pushdown systems. *Theor. Comput. Sci.* **549**, 127–145 (2014)
40. Thomas, W.: Automata on infinite objects. In: *Formal Models and Semantics*, pp. 133–191. Elsevier (1990)
41. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.* **47**(1), 6:1–6:45 (2014)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



ReVEAL: GNN-Guided Reverse Engineering for Formal Verification of Optimized Multipliers

Chen Chen^{1,4*}, Daniela Kaufmann², Chenhui Deng³, Zhan Song¹,
Hongce Zhang⁴, and Cunxi Yu¹

¹ University of Maryland, College Park, USA
{cchen099, zhansong, cunxiyu}@umd.edu

² TU Wien, Austria
daniela.kaufmann@tuwien.ac.at

³ NVIDIA

cdeng@nvidia.com

⁴ Hong Kong University of Science and Technology (Guangzhou)
hongcezh@hkust-gz.edu.cn

Abstract. We present REVEAL, a graph-learning-based method for reverse engineering of multiplier architectures to improve algebraic circuit verification techniques. Our framework leverages structural graph features and learning-driven inference to identify architecture patterns at scale, enabling robust handling of large optimized multipliers. We demonstrate applicability across diverse multiplier benchmarks and show improvements in scalability and accuracy compared to traditional rule-based approaches. The method integrates smoothly with existing verification flows and supports downstream algebraic proof strategies.

Keywords: Reverse engineering · Multiplier · Graph learning · Formal verification · Computer algebra

1 Introduction

Gate-level arithmetic circuits, in particular integer multipliers that have been optimized via logic synthesis, continue to pose significant challenges for current formal verification techniques. Despite notable advancements in computer algebra-based (CA) techniques [8, 14, 17, 21], these methods often rely on syntactic heuristics and hence struggle with synthesized circuits due to optimizations that obscure their original word-level structure [7, 17, 21, 32, 33].

Our goal is to verify such optimized circuits by reverse-mapping them back to their original non-optimized word-level representations that can be efficiently verified using algebraic reasoning. We ensure the correctness of this mapping through Boolean satisfiability (SAT)-based equivalence checking.

* This work was conducted during Chen Chen’s visit to the University of Maryland, College Park.

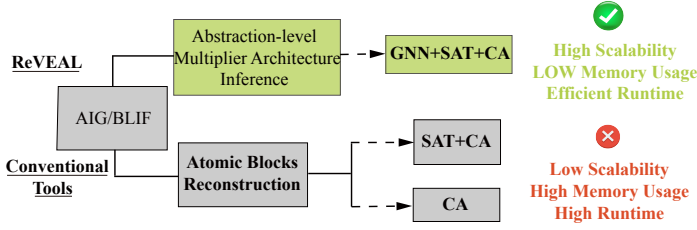


Fig. 1: Overview of ReVEAL versus Conventional Tools

Traditional computer algebra techniques reduce the task of circuit verification to an ideal membership test for the specification polynomial, typically utilizing Gröbner bases [5] to simplify the polynomials through backward rewriting. In [13,14], an incremental column-wise verification approach together with adder substitution is used in the tool AMULET2 to handle unoptimized multipliers. However, the verification of optimized multipliers still presents significant challenges due to blow-ups in the intermediate rewriting steps. In REVSCA2 [21] this is addressed by algebraic reverse engineering to recover word-level components such as full adders and half adders. Most recently, DYNPHASEORDEROPT [17] optimizes the encoding of the phases of occurring signals to enhance backward rewriting and keep the sizes of intermediate polynomials small. However, despite these advances, logic synthesis often applies aggressive optimizations that obscure the original circuit structure. As a result, many original atomic blocks cannot be restored through cut enumeration, which limits the effect of the aforementioned rewriting methods. Moreover, the inevitable occurrence of numerous OR chains and XOR gates in multiplier circuits further complicates polynomial rewriting, resulting in ongoing issues with monomial blow-ups.

Recent work by Li et al. [19] reconstructs an adder tree from optimized multipliers to build a structurally similar reference design, and then verifies it via CA and SAT-based equivalence checking. However, such reconstruction-based reverse engineering fundamentally depends on explicit structural pattern matching, which is fragile under aggressive synthesis and technology mapping. As a result, boundary recovery can become expensive or fail entirely, and the approach is often architecture-specific (e.g., tailored to adder trees), making it difficult to handle diverse designs such as 4-to-2-compressor-based and counter-based Wallace trees, XOR-heavy addition structures, or Booth-encoded multipliers where PPG/PPA boundaries are blurred.

Due to the above limitations of reconstruction-based methods [19], ReVEAL instead formulates the task as a learning-based inference problem. Traditional reverse engineering largely depends on recovering explicit boundaries and subgraph patterns, which can be broken or smeared by aggressive synthesis. In contrast, GNNs can aggregate information over neighborhoods via message passing and encode both local motifs and broader connectivity trends, allowing ReVEAL to capture more optimization-invariant cues from the netlist graph and thus remain effective even when canonical structures are not directly recoverable.

Our Contributions. In this work, we propose a novel domain-specific GNN-based reverse engineering approach for architectural identification and demonstrate its significant impact in orthogonally accelerating computer algebra-based formal verification techniques by addressing the aforementioned challenges.

The contributions of this paper are as follows:

- To the best of our knowledge, ReVEAL is the first framework to integrate machine learning (ML) techniques into an end-to-end formal verification pipeline for multiplier circuits, demonstrating significant improvements in both runtime and scalability.
- We present a novel approach that combines cone extraction with word-level functional and structural features in GNNs. This method enables the model to focus on small graphs within the critical cone, avoiding the performance degradation and long training times typically associated with large graphs.
- We compare ReVEAL with advanced (SAT+)CA-based verification tools. The results show that ReVEAL achieves a $4.90\times$ speedup and a $19.97\times$ reduction in memory usage compared to state-of-the-art CA tools. Additionally, ReVEAL outperforms SAT+CA methods with a $13.39\times$ speedup.

2 Preliminaries

In this section we provide background information on different multiplier architectures (Section 2.1), a short introduction to GNNs (Section 2.2), a very brief discussion of SAT solvers and SAT sweeping (Section 2.3), as well as the problem formulation (Section 2.4).

2.1 Multiplier Architectures

Multipliers typically involve three stages: (1) *Partial Product Generator (PPG)*, (2) *Partial Product Accumulator (PPA)*, and (3) *Final Stage Adder (FSA)*.

The PPG can employ either a simple approach that pairs corresponding bits of the multiplicands to generate each partial product using logical conjunction or a Booth encoding scheme to process multiple bits at a time.

The PPA then sums these partial products using structures like Wallace trees, Dadda trees, or 4-to-2 compressor trees, which reduce gate delays and minimize sequential additions compared to traditional array accumulators.

Finally, the FSA combines the last two rows of partial products using adders, which can be broadly categorized into non-tree-based and tree-based adders (e.g., Brent–Kung, Kogge–Stone, Sklansky, Han–Carlson), with tree adders typically showing more regular, highly connected prefix structures. We list the most common multiplier architectures in Table 1.

2.2 Graph Neural Networks

We refer readers to [10] for a comprehensive introduction to graph representation learning and GNNs. A GNN encodes a graph $G = (V, E)$ into a more compact

Table 1: Categorization of Multiplier Architectures

Stage	Types	
① PPG	• Booth Encoding	• Simple
② PPA	• Simple Array	• Wallace Tree
	• Dadda Tree	• 4-to-2 Compressor Tree
	• Counter-based Wallace Tree	
③ FSA	• Ripple Carry Adder	• Carry Look-Ahead Adder
	• Carry Skip Adder	• Serial Prefix Adder
	• Brent-Kung Adder	• Sklansky Adder
	• Han-Carlson Adder	• Ladner-Fischer Adder
	• Kogge-Stone Adder	

Note: Tree adders are marked using blue cell colors.

representation. Starting with initial node features $\{\mathbf{x}_i^{(0)} \mid i \in V\}$, GNNs iteratively update hidden state vectors $\mathbf{x}_i^{(l)}$ using:

$$\mathbf{m}_i^{(l)} = \text{AGGREGATE}^{(l)} \left(\{\mathbf{x}_j^{(l-1)} \mid j \in \mathcal{N}(i)\} \right), \quad (1)$$

$$\mathbf{x}_i^{(l)} = \text{COMBINE}^{(l)} \left(\mathbf{m}_i^{(l)}, \mathbf{x}_i^{(l-1)} \right), \quad (2)$$

where $\mathcal{N}(i)$ are the one-hop neighbors of node i . The AGGREGATE function (e.g., mean or max pooling) pools neighbor features, and the COMBINE function integrates this with the node’s current feature. After L layers, the set of vectors $\{\mathbf{x}_i^{(L)} \mid i \in V\}$ capture the structural and feature information of the graph. A readout function then aggregates these into a graph embedding for classification, enabling GNNs to effectively model complex dependencies in circuit designs for various EDA applications.

Building on this capability, recent research has explored the use of GNNs for reverse engineering tasks. HOGA [9] and GAMORA [31] leverage gate-level Boolean reasoning to identify adders at the register transfer level (RTL), while the DEEPGATE family [25] utilizes Boolean circuit representation learning to facilitate SAT solving. Furthermore, FGNN [11, 29] employs graph learning techniques to identify adder blocks and classify circuits.

These prior works studied the use of GNNs in low-level atomic block identification/reconstruction, while our proposed approach ReVEAL is the first work that investigates the application of GNNs in the inference of high-level design choices (abstraction-level architectural inference).

2.3 Modern SAT Solvers and SAT Sweeping

KISSAT [3, 4] is a state-of-the-art SAT solver, which employs clausal congruence closure to efficiently handle large-scale equivalence checking problems, particularly for circuits containing numerous small isomorphic subcircuits.

SAT sweeping [35] is an efficient equivalence checking approach that combines simulation with incremental SAT solving. It detects equivalent nodes through

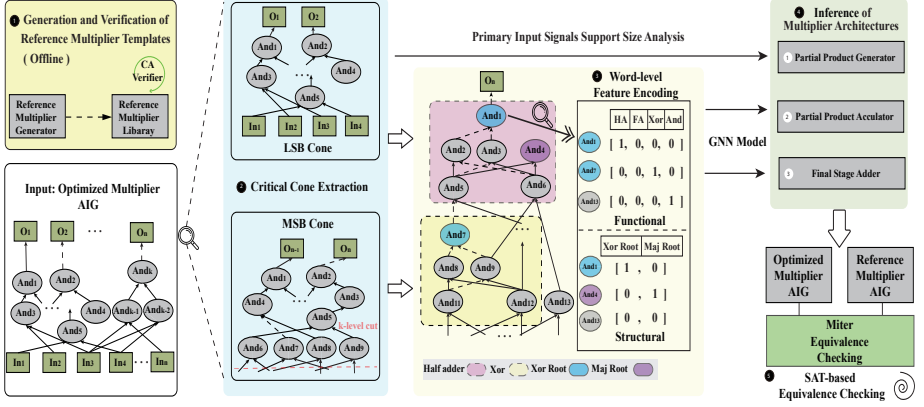


Fig. 2: The Workflow of ReVEAL

simulation, constructs intermediate miters, and runs SAT solving in topological order on two circuits to iteratively refine and merge equivalent nodes.

2.4 Problem Formulation

We denote an optimized multiplier gate-level netlist in AIG form as G with bit-width N . Table 1 summarizes the candidate structures for the three stages considered in this work. Given G , ReVEAL infers the corresponding (PPG, PPA, FSA) architecture choices from Table 1. Based on the inferred architectures and N , we select the matching pre-verified template from our library and use SAT-based combinational equivalence checking (CEC) to validate functional equivalence between G and the selected template.

3 Methodology – the ReVEAL Framework

Figure 2 presents the workflow of our reverse engineering framework **ReVEAL**, which consists of five main steps. Steps 2–4 are explained in more detail in the following sections.

1 Offline Generation and Verification of Reference Multiplier Templates. We generate RTL templates for multipliers of various bitwidths and architectures using multiplier generators [20] [27]. These templates are processed with Yosys [30] to produce their corresponding And-Inverter Graphs (AIGs). All templates are formally certified using AMULET2 [14] and are collected in our *reference multiplier library*. Notably, the generation and certification processes are performed offline.

2 Critical Cone Extraction. The input to ReVEAL is a gate-level multiplier $\mathcal{G}$, optimized by logic synthesis tools, such as ABC [22]. We extract the Least Significant Bit (LSB) and Most Significant Bit (MSB) cones of $\mathcal{G}$, which

allows us to perform architecture inference on smaller subcircuits and hence enhances the generalization capability of the GNN model.

③ Word-level Feature Encoding. For the extracted LSB and MSB cones, we perform reverse identification of arithmetic blocks such as full adders and half adders to encode word-level features on $\mathcal{G}$.

④ Inference of Multiplier Architecture. By combining graph analysis on primary input signals and GNN model inference, we predict the architecture of the optimized multiplier, identifying its PPG, PPA, FSA.

⑤ SAT-based Equivalence Checking. Using the generated reference multiplier library of step 1, we locate a pre-verified template with the same architecture as the prediction of the optimized multiplier. A miter circuit is constructed by combining the optimized multiplier and the reference template for equivalence checking. We verify the miter using SAT solving/sweeping; see **Q2** in the experimental evaluation.

3.1 Critical Cone Extraction

Previously, GNNs struggled with long-range dependencies in large-scale graphs, leading to over-smoothing [18] and over-squashing [2]. Over-smoothing makes node representations indistinct, while over-squashing compresses excessive information into fixed-size embeddings, causing loss of critical details. These issues hinder the performance and scalability of GNNs on large graph structures.

To tackle these challenges, we partition the graph to concentrate on critical cones relevant to their corresponding primary stages of the multiplier architecture: PPG, PPA, FSA, see Figure 3 for an example. Firstly we handle the **①** PPG and **②** PPA stages by pre-processing the input AIG and extract the C least significant primary outputs and their corresponding input cones. In our experimental dataset (see Section 4), which starts with a minimum input bit-width of 32, we have empirically observed that the cones of the $C = 8$ LSBs contain sufficient distinct information to successfully identify the **②** PPA, without being affected by contamination from the **③** FSA stages. Moreover, as the bit-width increases, the cones of the first 8 LSBs trace back to 16 input bits, displaying highly similar cone structures, which are beneficial for GNNs generalization.

Regarding the **①** PPG, we directly perform graph analysis on the cone extracted from the LSBs. For doing so, we define the support size $\text{SuppSize}(X)$ for a node X as:

$$\text{SuppSize}(X) = |\{I \in \{\text{Primary Inputs}\} : \exists I \rightsquigarrow X\}|,$$

where $\exists I \rightsquigarrow X$ indicates that there exists a directed path from primary input node I to node X . It recursively traverses its fan-in nodes, incrementally counting the primary inputs that influence the outputs.

We extract a graph slice where the support size of the primary input signals is 2. Since Booth multipliers involve multiple input signals for each product generator, by analyzing the fanout of primary inputs and the logic levels within the slice, we can determine the use of Booth encoding. Specifically, when the

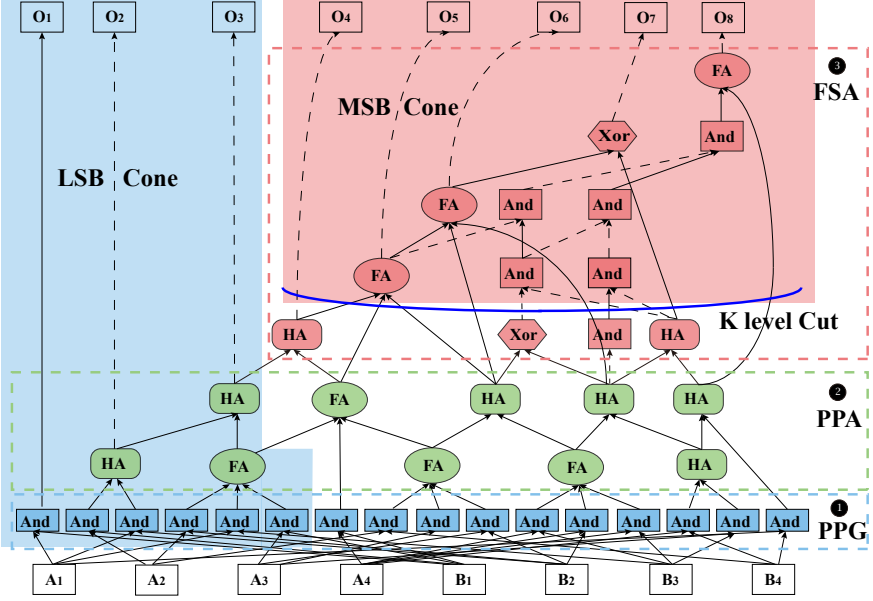


Fig. 3: Example of LSB & MSB cone extraction for a 4-bit multiplier

supporting size is set to 2 and the slice has more than 2 logic levels, it indicates that Booth encoding is being utilized. Similarly, we can also identify higher-radix multipliers by examining the output support nodes of the primary inputs, as Booth- n encoding, $n + 1$ bits of the multiplier are examined, and n multiplier bits are encoded.

For the ③ FSA stage, we begin at the MSB, denoted as M , and extract the cone spanning from the M -th bit to the $(M - L)$ -th bit, where $L = 8$. This value is again empirically derived. To further isolate this cone from potential interference by the PPA and PPG stages, we implement a K -level cut of the MSB cone. The selection of K is contingent upon the multiplier bit-width N . Specifically, through empirical observation and symbolic regression on our experimental dataset, we have established a relationship between N and the corresponding logic level K required for effective graph partitioning; that is: $K = 5 + 2 \cdot \lfloor \log_2(N) \rfloor$. This formula ensures that for varying bit-widths—from 32 to 256 bits—the K -level cut adapts appropriately, maintaining consistent cone sizes and effectively focusing only on small cones relevant to the FSA stages. Combining the dynamic K -level cut with a fixed $L = 8$ ensures that the extracted cone retains critical structural and functional information pertinent to the FSA, while keeping the cone size within a manageable limit of a few hundred nodes. This size restriction is crucial for enhancing the training speed, allowing the learning process to remain efficient and maintain generalization during inference at larger bit widths.

Table 2: Feature Embeddings for Graph Neural Network Training

Level	Type	Feature	Value Type
Node	Basic	• Input edge1 inversion	0/1
		• Input edge2 inversion	0/1
		• PI/PO/inner node	0/1
	Function	• HA node	0/1
		• FA node	0/1
		• Remaining XOR node	0/1
		• AND node	0/1
	Structure	• MAJ root	0/1
		• XOR root	0/1
Graph	Structure	• Input count	Integer
		• Gate count	Integer
		• Graph density	Float
		• Clustering Coefficient	Float

3.2 Word-Level Feature Encoding

Feature representation is key to effective message propagation in GNNs, as it determines how information is transmitted and aggregated across the graph.

Table 2 summarizes our exploration of node-level and graph-level features. In addition to retaining the basic structural feature encoding used in HOGA and GAMORA, which employs a one-hot feature vector to represent node types such as primary inputs (PI), internal nodes, and primary outputs (PO), as well as inversions on edge, we enhance basic functional node features to enable reverse mapping to word-level atomic blocks [34].

For functional atomic block identification, we employ cut enumeration [23] in the graph $\mathcal{G}$ to reverse-engineer the functional components at the RTL, identifying elements such as full adders (FA) and half adders (HA), and remaining XOR gates that are not part of FA/HAs. For readers interested in the details of identification, we provide the complete procedure in Appendix A. By detecting all XOR3 and MAJ3⁵ combinations in the FAs, we pinpoint the corresponding input nodes (IN_1, IN_2, IN_3) and output nodes ($XOR3\ Root, MAJ3\ Root$). The same method applies to HAs. Through recursive traversal of the input and output nodes in each pair, we identify all AIG nodes that can be reverse-mapped to HAs and FAs. The remaining nodes that can be reverse-mapped to an XOR are annotated as XOR nodes, while the remaining AIG nodes are annotated as AND gates. In the examples shown in Figure 2, nodes such as AND1, AND7, and AND13 are each represented by a vector that indicates their respective functional blocks as HAs, FAs, XOR, and AND.

⁵ XOR3 refers to the logic gate implementing the three-input parity function $x \oplus y \oplus z$; MAJ3 denotes the three-input majority function $\text{maj}(x, y, z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$.

Cone extraction, as discussed in Section 3.1, enables us to work with smaller graphs, allowing for the quick computation of graph-level features such as graph density⁶, clustering coefficient⁷, cut input count, and gate count.

These features are crucial for identifying the sparsity of connections and cut fan-in numbers in different adder structures. Additionally, the time required for generating these functional and structural node features is negligible. This efficiency in feature computation significantly enhances our model’s ability to classify different types of adders while maintaining scalability in larger designs.

3.3 Inference of Multiplier Architecture

We use a multi-task hierarchical classification strategy built upon a bidirectional GraphSAGE [28] architecture to classify and more accurately distinguish different architectures of PPAs and FSAs. Recall, that we only require inference for PPA and FSA, as PPG can already be classified during critical cone extraction.

Directional GraphSAGE Model. The model captures directional dependencies through parallel message passing, where the graph neural network aggregates features from both incoming (predecessor) and outgoing (successor) edges. The directional embedding computation at layer k is defined as:

$$\begin{aligned} \mathbf{h}_v^{\text{pre}(k)} &= \text{ReLU} \left(\mathbf{W}_{\text{pre}}^{(k)} \cdot \text{MEAN} \left(\{ \mathbf{h}_u^{(k-1)} \mid u \in \mathcal{N}_{\text{in}}(v) \} \right) \right) \\ \mathbf{h}_v^{\text{suc}(k)} &= \text{ReLU} \left(\mathbf{W}_{\text{suc}}^{(k)} \cdot \text{MEAN} \left(\{ \mathbf{h}_u^{(k-1)} \mid u \in \mathcal{N}_{\text{out}}(v) \} \right) \right) \end{aligned} \quad (3)$$

with $\mathbf{W}_{\text{pre}}^{(k)}$ and $\mathbf{W}_{\text{suc}}^{(k)}$ being directional message parsing weights (which are trainable) for layer k , and $\mathcal{N}_{\text{in}}(v)$, $\mathcal{N}_{\text{out}}(v)$ being the one-hop neighbors of node v (following either incoming or outgoing edges).

After three message-passing layers, we compute normalized node embeddings through concatenation ($\parallel$) and batch normalization (BN) as follows:

$$\mathbf{h}_v^{\text{final}} = \text{BN} \left(\mathbf{h}_v^{\text{pre}(3)} \parallel \mathbf{h}_v^{\text{suc}(3)} \right) \quad \forall v \in G \quad (4)$$

and then obtain the graph-level representation via global mean pooling:

$$\mathbf{h}_G = \frac{1}{|G|} \sum_{v \in G} \mathbf{h}_v^{\text{final}} \quad (5)$$

⁶ Graph density: fraction of present edges out of all possible ones (value range: 0–1; higher means more connections).

⁷ Clustering coefficient: how tightly a node’s neighbors connect to each other (higher means a tighter group).

Hierarchical Classification. Our two-stage hierarchical design addresses the error accumulation problem in a direct 9-class FSA classification (cf. Table 1) through structural decomposition. The key insight stems from the fundamental architectural differences between tree-based and non-tree-based adders: tree structures exhibit regular logarithmic-depth patterns, while non-tree implementations often contain irregular subgraph motifs. This hierarchical strategy first performs a coarse-grained binary classification to distinguish tree-based and non-tree-based adders, followed by fine-grained classification within each category to identify specific adder types. The classification pipeline proceeds as follows:

Stage 1 - Tree vs. Non-tree Discrimination

$$\hat{y}_{\text{topo}} = \arg \max (\mathbf{W}_{\text{topo}} [\mathbf{h}_G \parallel f_{\text{level}}]) \quad (6)$$

where f_{level} denotes the maximum logic depth of the optimized circuit, capturing critical structural information. This stage effectively separates tree adders, characterized by deep recursive structures, from non-tree adders, which exhibit shallow combinational designs.

Stage 2 - Fine-grained Classification Conditioned on the predicted topology (tree vs. non-tree), we perform subtype classification by combining the graph embedding with lightweight structural summaries:

$$\hat{y}_{\text{sub}} = \arg \max (\mathbf{W}_t [\mathbf{h}_G \parallel f_{\text{fan}} \parallel \mathbf{f}_{\text{graph}}]), \quad t \in \{\text{nt}, \text{tree}\}. \quad (7)$$

Here, f_{fan} is the fan-in statistic extracted from the K -level MSB cone, and $\mathbf{f}_{\text{graph}} \in \mathbb{R}^3$ summarizes global structure (density, clustering coefficient, and average degree).

Multi-task Training Strategy. The model’s trainable parameters Θ comprise three components: directional message passing weights $\mathbf{W}_{\text{pre}}^{(k)}$ and $\mathbf{W}_{\text{suc}}^{(k)}$ from Eq. 3, where $k \in \{1, 2, 3\}$ represents the three layers of message passing, along with classifier weights $\mathbf{W}_{\text{topo}}$ for topology prediction, $\mathbf{W}_{\text{nt}}$ for non-tree subtypes, and $\mathbf{W}_{\text{tree}}$ for tree subtypes. The gradient composition rule combines task-specific gradients through conditional routing:

$$\frac{\partial \mathcal{L}}{\partial \Theta} = \frac{\partial \mathcal{L}_{\text{topo}}}{\partial \Theta} + \alpha \left[I(\hat{y}_{\text{topo}} = 0) \frac{\partial \mathcal{L}_{\text{nt}}}{\partial \Theta} + I(\hat{y}_{\text{topo}} = 1) \frac{\partial \mathcal{L}_{\text{tree}}}{\partial \Theta} \right], \quad (8)$$

where $I(\cdot)$ implements conditional gradient routing based on topology prediction $\hat{y}_{\text{topo}}$, with $\mathcal{L}_{\text{topo}}$ denoting binary cross-entropy loss, $\mathcal{L}_{\text{nt}}$ for 4-class non-tree classification, and $\mathcal{L}_{\text{tree}}$ for 5-class tree classification. The balancing coefficient $\alpha = 3.2$ is calibrated during initial warm-up training through gradient variance analysis. This process computes the Frobenius norms’ squared magnitudes $\|\nabla_{\Theta} \mathcal{L}_{\text{topo}}\|_F^2$ and $\|\nabla_{\Theta} \mathcal{L}_{\text{sub}}\|_F^2$ over 100 iterations, where $\mathcal{L}_{\text{sub}} = \mathcal{L}_{\text{nt}} + \mathcal{L}_{\text{tree}}$.

Statistical aggregation calculates the running averages $\mathbb{E}[\cdot] = \frac{1}{100} \sum_{t=1}^{100} (\cdot)_t$, determining the optimal coefficient α via:

$$\alpha = \sqrt{\frac{\mathbb{E}[\|\nabla_{\Theta} \mathcal{L}_{\text{topo}}\|_F^2]}{\mathbb{E}[\|\nabla_{\Theta} \mathcal{L}_{\text{sub}}\|_F^2]}} \approx 3.2, \quad (9)$$

where $\|\cdot\|_F$ denotes the Frobenius norm, and $\|\cdot\|_F^2$ represents its squared magnitude, which is used here to measure the gradient strength for balancing tasks.

Parameter updates follow distinct pathways: message passing weights $\mathbf{W}_{\text{pre/suc}}$ receive combined gradients from both tasks, while $\mathbf{W}_{\text{topo}}$ updates exclusively through $\mathcal{L}_{\text{topo}}$. The subtype classifiers $\mathbf{W}_{\text{nt}}$ and $\mathbf{W}_{\text{tree}}$ activate only when $\hat{y}_{\text{topo}}$ predicts non-tree (0) or tree (1) topologies, respectively. The unified update rule preserves batch-normalized stability (Eq. 4):

$$\Theta^{(t+1)} = \Theta^{(t)} - \eta (\nabla_{\Theta} \mathcal{L}_{\text{topo}} + \alpha \nabla_{\Theta} \mathcal{L}_{\text{sub}}). \quad (10)$$

The proposed hierarchical classification framework, coupled with the multi-task training strategy, effectively mitigates error accumulation and ensures robust performance by leveraging topology-aware feature decomposition and gradient-balanced optimization.

4 Experiments

Experimental Setup. All experiments are run on a server running Ubuntu 20.04.4 LTS, with Intel(R) Xeon(R) Gold 6418H processors and 64 GB of memory. We set a timeout (TO) of 3600 seconds and a memory limit (MO) of 8 GB.

All multiplier circuits are converted to AIG format using Yosys [30] and ABC [22]. All generated reference multipliers are verified using AMULET2 [14] with 64 pre-prepared templates per bit-width (step 1). The algorithms for critical cone extraction (step 2) and word-level feature encoding (step 3) are implemented and integrated in ABC. For miter equivalence checking (step 5) we consider the SAT solver KISSAT 4.0.1 [3] and ABC’s SAT-sweeping-based combinational equivalence checkers `&fraig`, `cec` and `&cec`; the rationale for this solver combination is analyzed in **Q2**.

Dataset. We use the multipliers generated by the generators Multgen [27] and GenMul [20]⁸. GenMul provides 28 distinct architectures, while Multgen supports 36 architectures. We include all available architectures with one exception: for Multgen, we exclude Booth-encoding templates at bit-widths of 4 and above, because these instances cannot be correctly parsed by Yosys during SystemVerilog-to-AIG conversion.

⁸ We do not consider the richer Aoki benchmark set [12] in our analysis due to the discontinuation of their online generator, which renders it impossible to obtain the small-width cases necessary for model training.

Table 3: Inference Accuracy for Each Stage

Stage	TOP 1*	TOP 2*	TOP 3*
Stage PPG	100%	–	–
Stage PPA	100%	–	–
Stage FSA	85%	97%	98%

* TOP 1, TOP 2, and TOP 3 represent the prediction accuracy when the ground truth is among the model’s top 1, top 2, and top 3 ranked predictions, respectively.

To build the training and evaluation sets, we use even-numbered bit-widths from 32 to 60 bits for training and evaluate on unseen 64-, 128-, and 256-bit designs. Following prior work [17, 21], all optimized multipliers are produced using ABC operators: **dc2** or **resyn3**. All selected designs—both original and optimized—are normalized to AIG using Yosys [30] and ABC [22], ensuring consistent representation across generators, optimizations, and bit-widths. This design enables a rigorous assessment of the model’s ability to generalize to unseen, larger bit-widths, beyond the training regimes.

Machine Learning Configuration. We train our model for 200 epochs per optimization operator. All hyperparameters (e.g., learning rate, batch size, and weight decay) are automatically tuned using Optuna [1]; the final configurations and trained model checkpoints are provided in our open-source repository. Our pipeline demonstrates exceptional efficiency: cone extraction plus word-level feature extraction complete in under 1 second per case. A complete training cycle consists of 329 seconds for feature extraction and 321 seconds for training, totaling 650 seconds. Our training scales well as bit-widths grow thanks to critical cone extraction. In contrast, conventional full-graph training approaches, which perform equivalent computations on full AIGs without cone extraction, are 182× slower.

Research Questions. We organize the presentation of experimental results to address the following research questions:

Q1: *Can GNNs and analytical techniques demonstrate robust performance and generalization capability for architectural inference across optimized netlists?*

The inference accuracy is shown in Table 3. We also summarize, for the 64-bit dc2-optimized case, all top-3 predictions in Appendix B. In *Stage PPA* we observe that the LSBs cone maintains similar cone graph sizes and structural similarities across different bit-widths, enhancing the model’s learning capability for larger bit-widths. As a result, our training process achieves an accuracy rate of 100% for the first two stages. Furthermore, our model demonstrates high accuracy in *Stage FSA*, with the top three predictions nearly achieving complete correctness. We observe that when the circuits have been optimized using **dc2**, the FSA prediction errors mainly appear on Brent–Kung (BK) cases. This happens because our MSB-centered cone extraction is only an approximate par-

Table 4: Solver Performance for Miter Solving

Opt	dc2			resyn3		
	Solved	Avg Time (s)	#. fastest	Solved	Avg Time (s)	#. fastest
cec	60/64	692.57	0	60/64	1007.20	1
&cec	64/64	8.50	46	64/64	6.83	50
&fraig -y	36/64	1848.53	12	39/64	1877.33	8
&fraig -x	37/64	1765.14	0	39/64	1726.71	5
kissat	64/64	96.79	6	64/64	186.55	0

tition. In some extensively optimized cases, we observe that structurally similar but not identical templates of the FSA can facilitate faster equivalence checking. Therefore, we retain the top three predicted templates and have them participate in SAT solving in parallel.

Q2: *Are modern SAT solvers and SAT sweeping techniques efficient for equivalence checking between the optimized and the golden multiplier?*

We selected the SAT-sweeping tools mentioned in the experimental setup for equivalence checking of optimized (synthesized) and original (golden) circuits. We tested 64-bit multipliers with 64 different templates. Our quantitative analysis in Table 4 reveals that while **&cec** achieves the highest success rate, **KISSAT**, **&fraig -y** and **&cec** remain essential for handling specific corner cases. Notably, the SAT-sweeping approach demonstrates remarkable memory efficiency compared to computer algebra-based methods — a critical advantage when verifying complex arithmetic circuits, which we will quantitatively validate in terms of memory consumption later for **Q3**. Based on these observations, we implement a hybrid parallel verification framework that concurrently executes three strategically selected solvers (**&fraig -y**, **&cec**, and **KISSAT**), terminating the verification process immediately when any solver returns a conclusive result.

Q3: *Is architecture identification helpful for computer-algebra-based multiplier verification?*

Comparison with State-of-the-Art CA Verification Tools. We systematically test all 64 possible template combinations and successfully verify equivalence between the top three predicted templates and the circuit under test. For a fair comparison as in [17, 21], we use the same optimizations, **dc2** and **resyn3** in **ABC**. Experimental results in Table 5 and Figure 4 show that **ReVEAL** successfully solves nearly all benchmarks, except one limited by excessive SAT-solving time. In general, minor errors in the FSA stage prediction are tolerable—SAT solving and sweeping remain highly effective even across structurally different FSA templates, providing a robustness cushion against FSA misclassifications.

Specifically, we highlight the comparison of verification times and memory usage across various tools in Figure 4. We include per-case solving details in Appendix C. In general, existing CA tools suffer from vanishing monomial computations, which make them inefficient for handling large optimized circuits. On

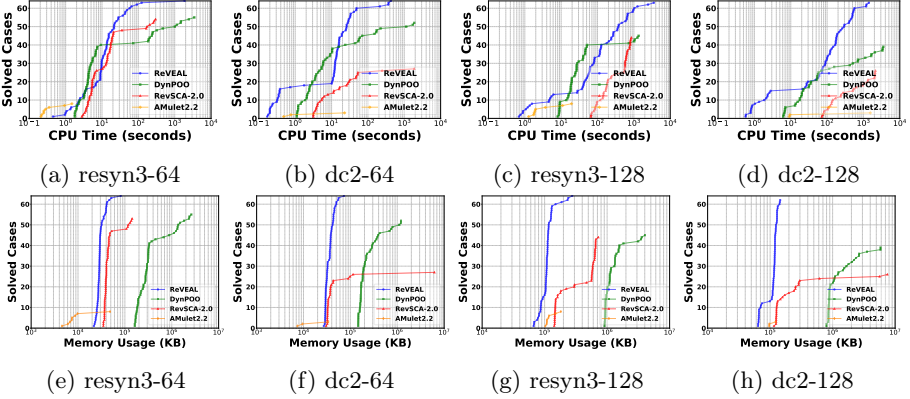


Fig. 4: Runtime and memory usage across different “optimization-bitwidths”.

average, our method runs in 174.34 seconds using 75.01 MB memory, while DYN-PHASEORDEROPT [17] requires 936.40 seconds and 1,374.40 MB. This demonstrates a $5.37\times$ speed improvement and $18.33\times$ memory reduction in favor of our approach. As can be seen in Table 5, the tool REVSCA2 reports correct circuits as “buggy”. This is because REVSCA2 has bugs in reversely identifying functional arithmetic blocks, which cause the circuit to become inequivalent to the original circuit.

Impact of Misidentification on Verification Runtime. As an illustration, we run the following experiment in Table 6. We fix a 64-bit test multiplier that is optimized with resyn3 (column “Test circuit”) and use equivalence checking to compare it against multipliers of different architectures. In Table 6(a), we resemble the situation when FSA (stage 3) is mispredicted, and Table 6(b) mimics the case when PPA (stage 2) is mispredicted. Note that not all FSA misprediction results in long runtime (see case 6 and 7). Despite that PPA misprediction leads to excessively long solving time, as shown by the earlier experiment in Table 3, our model achieves a high accuracy in PPG and PPA prediction and this helps avoid the long equivalence checking time.

Buggy-circuit robustness. We also evaluate REVEAL on buggy designs. For each optimized 64-bit AIG, we create buggy variants by rewiring the fanins of three internal nodes (preserving acyclicity) and resample until random simulation finds a mismatch at some primary output. We then run SAT-based CEC against the corresponding template. ABC’s cec takes 0.05 s on average (max 1.98 s), and all instances are SAT (non-equivalence detected).

Certification. Once REVEAL concludes equivalence, we can provide proof certificates for both verification steps: (i) each unoptimized template in our library is certified by AMULET2 via an algebraic proof certificate (a sequence of polynomial additions and multiplications) [15]; and (ii) SAT-based CEC using KISSAT

Table 5: The number of solvable cases for different tools across benchmarks

Tool	Result	Benchmark			
		64bit		128bit	
		dc2	resyn3	dc2	resyn3
AMulet-2.2	Solved	3/64	8/64	3/64	8/64
	TO	61	55	61	56
	MO	0	1	0	0
	“Circuit buggy”	0	0	0	0
RevSCA-2.0	Solved	26/64	54/64	26/64	44/64
	TO	7	3	7	8
	MO	21	7	29	7
	“Circuit buggy”	2	0	2	5
DynPOO	Solved	52/64	55/64	39/64	45/64
	TO	12	9	22	7
	MO	0	0	3	12
	“Circuit buggy”	0	0	0	0
ReVEAL	Solved	64/64	64/64	64/64	63/64
	TO	0	0	0	1
	MO	0	0	0	0
	“Circuit buggy”	0	0	0	0
	Stage FSA acc. (%)	96.88	100.00	92.19	100.00

Table 6: Effect of missclassifications.

#	Test circuit	Template	Runtime
1	SP_AR_BK	SP_AR_BK	3.04
2	SP_AR_BK	SP_AR_CK	3.37
3	SP_AR_BK	SP_AR_CL	5083.42
4	SP_AR_BK	SP_AR_KS	232.18
5	SP_AR_BK	SP_AR_LF	33.46
6	SP_AR_BK	SP_AR_RC	3.37
7	SP_AR_BK	SP_AR_SE	3.15

(a) Missclassified FSA.

#	Test circuit	Template	Runtime
1	SP_DT_BK	SP_DT_BK	3.8
2	SP_DT_BK	SP_AR_BK	TO
3	SP_DT_BK	SP_CWT_BK	TO
4	SP_DT_BK	SP_WT_BK	TO

(b) Missclassified PPA.

outputs a DRUP clausal proof certificate. Both certificate types can be checked by certified proof checkers [15, 26].

Comparison with CA+SAT Methods on Extensively Optimized Multipliers. In Table 7, we further evaluate extensively optimized⁹ multipliers, comparing ReVEAL with RefSCAT [19], a method that combines SAT and CA by reconstructing complete references through adder boundary detection using a constraint satisfaction algorithm. For training and testing, we use the same settings as above in Section 4. Training uses even bit-width, ranging from 32 to 60 bits and testing uses unseen 64, 128, and 256-bit designs. Since RefSCAT is not publicly available, we use the experimental data from [19] for comparison. In terms of runtime, ReVEAL outperforms RefSCAT by a factor of 13.39, despite one corner

⁹ The multipliers are optimized using the following sequence in ABC, applied twice: `&mfs: logic; mfs2 -W 20; mfs; st; dc2 -1; resub -1 -K 16 -N 3 -w 100; logic; mfs2 -W 20; mfs; st; resyn -1; resyn; resyn2; resyn3; dc2 -1`

Table 7: Time Comparison between RefSCAT and ReVEAL.

Benchmark	RefSCAT	ReVEAL					
	Time (s)	CE (s)	FE (s)	Inference (s)	EQ (s)	Total Time (s)	
128_128_U_SP_DT_HCA	2518.8	0.51	0.10	0.01	40.13	40.74	
128_128_U_SP_WT_HCA	1791.1	0.49	0.11	0.01	42.69	43.29	
128_128_U_SP_DT_CK	144.1	0.49	0.10	0.01	24.84	25.44	
128_128_U_SP_DT_KS	156.4	0.56	0.08	0.01	29.00	29.65	
128_128_U_SP_WT_CK	403.2	0.54	0.10	0.01	108.51	109.16	
128_128_U_SP_WT_KS	422.1	0.51	0.09	0.01	54.11	54.71	
128_128_U_SP_WT_LF	1889.2	0.49	0.11	0.01	41.21	41.83	
128_128_U_SP_DT_CL	127.2	0.50	0.08	0.01	211.07	211.66	

* CE: Cone Extraction; FE: Feature Extraction; EQ: Equivalence Checking;

case slowed down by SAT sweeping time. The results show that ReVEAL’s time overhead is almost entirely spent on equivalence checking, while the preprocessing phase takes less than 1 second. In contrast, RefSCAT uses significant time on adder boundary detection, particularly in highly optimized cases. Notably, RefSCAT currently cannot verify Booth-encoded or non-adder-based multipliers, requiring online RTL multiplier reconstruction that introduces RTL to AIG conversion overhead and online computer-algebraic verification runtime. Our approach, on the other hand, is faster and more versatile for reverse engineering and architectural inference.

Q4: *Are the proposed cone extraction, feature extraction, and neural network models domain-specifically helpful in the task?*

To evaluate the contribution of each technique in ReVEAL, we conduct an ablation study by selectively removing key components of our approach. The results are shown in Table 8. First, in *ReVEAL w/o cone extraction*, we train the model on the entire large graph without applying cone extraction. This leads to a significant performance decline during both *Stage PPA* and *Stage FSA* inference, primarily due to the over-squeezing and over-smoothing problems that GNNs face when operating on large graphs, demonstrating the necessity of cone extraction for efficient processing. Second, in *ReVEAL w/o word-level features*, we replace our word-level features with the basic functional AIG features used in HOGA and GAMORA. This results in a decrease in accuracy, indicating that word-level feature extraction contributes positively to the model’s performance. Last, we conduct a comprehensive comparison of our approach with existing mainstream *GNN models*. Our analysis reveals that models lacking hierarchical classification exhibited higher misclassification rates for both non-tree and tree adders. In contrast, ReVEAL, which integrates hierarchical classification with bidirectional GraphSAGE, significantly reduces misclassification and enhances overall classification accuracy.

Table 8: Ablation Study and Performance Comparison of Different Methods.

	Method	Stage	Accuracy (%)		
			TOP 1	TOP 2	TOP 3
Base	ReVEAL	Stage PPA	1.0	/	/
	ReVEAL	Stage FSA	0.85	0.97	0.98
w/o CE*	ReVEAL (w/o CE*)	Stage PPA	0.15	/	/
	ReVEAL (w/o CE*)	Stage FSA	0.16	0.28	0.40
w/o WF*	ReVEAL (w/o WF*)	Stage PPA	1.0	/	/
	ReVEAL (w/o WF*)	Stage FSA	0.76	0.94	0.95
GNN Model	GCN	Stage FSA	0.72	0.9	0.94
	GAT		0.75	0.91	0.95
	GraphSAGE		0.80	0.96	0.97
	SagePooling [16]		0.71	0.91	0.92
	HOGA [9]		0.79	0.92	0.94
	Graph Transformer [24]		0.64	0.86	0.91

* CE: Cone Extraction; WF: Word Feature Extraction

5 Conclusion and Future Work

In this paper, we have introduced ReVEAL, the first GNN-based framework for architectural reasoning in the reverse engineering of optimized multipliers. ReVEAL integrates critical cone extraction, word-level feature encoding, and a hierarchical multi-task classification model to achieve strong generalizability and high accuracy. Training scales well when bit-widths grow because critical cone extraction fixes the learning scope to small stage-specific subgraphs. By combining GNN-based reverse engineering with modern SAT solvers and computer algebra verifiers, ReVEAL effectively overcomes the performance bottlenecks inherent in traditional CA and CA+SAT-based verification methods for optimized multiplier verification. A future extension for the work includes applying the critical-cone learning paradigm to MAC units and other more complex datapath circuits.

Acknowledgments. This research was supported by U.S. National Science Foundation, under awards CCF-2229562, CCF-2403134, CCF-2349461, and CCF-2349670. This research was also supported by the Austrian Science Fund (FWF) [10.55776/ESP666].

Code and Data Availability Statement The source code that supports the contributions of this work is freely available at <https://github.com/chestercc1997/ReVEAL>. The complete dataset and reproducible models are available at <https://huggingface.co/datasets/SeaSkysz/reveal>. The authors confirm that the data supporting the findings of this study are available within the paper and the artifact.

Disclosure of Interests The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M.: Optuna: A next-generation hyperparameter optimization framework. In: The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 2623–2631 (2019)
2. Alon, U., Yahav, E.: On the bottleneck of graph neural networks and its practical implications. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3–7, 2021. OpenReview.net (2021), <https://openreview.net/forum?id=i80OPhOCVH2>
3. Biere, A., Fazekas, K., Fleury, M., Froleyks, N.: Clausal congruence closure. In: 27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024). Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2024)
4. Biere, A., Fazekas, K., Fleury, M., Froleyks, N.: Clausal equivalence sweeping. In: FMCAD 2024. pp. 236–241. TU Wien Academic Press (2024)
5. Buchberger, B.: Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal. Ph.D. thesis, University of Innsbruck (1965)
6. Chatterjee, S.: On algorithms for technology mapping. University of California, Berkeley (2007)
7. Ciesielski, M., Su, T., Yasin, A., Yu, C.: Understanding algebraic rewriting for arithmetic circuit verification: a bit-flow model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **39**(6), 1346–1357 (2019)
8. Ciesielski, M., Yu, C., Brown, W., Liu, D., Rossi, A.: Verification of gate-level arithmetic circuits by function extraction. In: Proceedings of the 52nd Annual Design Automation Conference. pp. 1–6 (2015)
9. Deng, C., Yue, Z., Yu, C., Sarar, G., Carey, R., Jain, R., Zhang, Z.: Less is more: Hop-wise graph attention for scalable and generalizable learning on circuits. In: Proceedings of the 61st ACM/IEEE Design Automation Conference. pp. 1–6 (2024)
10. Hamilton, W.L.: Graph representation learning. Morgan & Claypool Publishers (2020)
11. He, Z., Wang, Z., Bai, C., Yang, H., Yu, B.: Graph learning-based arithmetic block identification. In: 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD). pp. 1–8. IEEE (2021)
12. Homma, N., Watanabe, Y., Aoki, T., Higuchi, T.: Formal Design of Arithmetic Circuits Based on Arithmetic Description Language. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* **89-A**(12), 3500–3509 (2006). <https://doi.org/10.1093/IETFEC/E89-A.12.3500>
13. Kaufmann, D., Biere, A.: Amulet 2.0 for verifying multiplier circuits. In: TACAS (2). LNCS, vol. 12652, pp. 357–364. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_19
14. Kaufmann, D., Biere, A.: Improving amulet2 for verifying multiplier circuits using sat solving and computer algebra. *International Journal on Software Tools for Technology Transfer* **25**(2), 133–144 (2023)
15. Kaufmann, D., Fleury, M., Biere, A., Kauers, M.: Practical algebraic calculus and Nullstellensatz with the checkers Pacheck and Pastèque and Nuss-Checker. *Formal Methods in System Design* (2022). <https://doi.org/10.1007/s10703-022-00391-x>
16. Knyazev, B., Taylor, G.W., Amer, M.R.: Understanding attention and generalization in graph neural networks. In: Wallach, H.M., Larochelle, H., Beygelzimer, A., d’Alché-Buc, F., Fox, E.B., Garnett, R. (eds.) *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information*

- Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada. pp. 4204–4214 (2019), <https://proceedings.neurips.cc/paper/2019/hash/4c5bfcfc8584af0d967f1ab10179ca4b-Abstract.html>
17. Konrad, A., Scholl, C.: Symbolic computer algebra for multipliers revisited-it’s all about orders and phases. In: FMCAD 2024. pp. 261–271. TU Wien Academic Press (2024)
 18. Li, Q., Han, Z., Wu, X.: Deeper insights into graph convolutional networks for semi-supervised learning. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018. pp. 3538–3545. AAAI Press (2018). <https://doi.org/10.1609/AAAI.V32I1.11604>, <https://doi.org/10.1609/aaai.v32i1.11604>
 19. Li, R., Li, L., Yu, H., Fujita, M., Jiang, W., Ha, Y.: Refscat: Formal verification of logic-optimized multipliers via automated reference multiplier generation and scasat synergy. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024)
 20. Mahzoon, A., Große, D., Drechsler, R.: Genmul: Generating architecturally complex multipliers to challenge formal verification tools. In: Recent Findings in Boolean Techniques: Selected Papers from the 14th International Workshop on Boolean Problems (2021), <https://api.semanticscholar.org/CorpusID:235831211>
 21. Mahzoon, A., Große, D., Drechsler, R.: Revsca-2.0: Sca-based formal verification of nontrivial multipliers using reverse engineering and local vanishing removal. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **41**(5), 1573–1586 (2021)
 22. Mishchenko, A., Chatterjee, S., Brayton, R.K.: Dag-aware AIG rewriting a fresh look at combinational logic synthesis. In: Sentovich, E. (ed.) Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006. pp. 532–535. ACM (2006). <https://doi.org/10.1145/1146909.1147048>, <https://doi.org/10.1145/1146909.1147048>
 23. Pan, P., Lin, C.: A new retiming-based technology mapping algorithm for lut-based fpgas. In: Cong, J., Kaptanoglu, S. (eds.) Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, FPGA 1998, Monterey, CA, USA, February 22-24, 1998. pp. 35–42. ACM (1998). <https://doi.org/10.1145/275107.275118>, <https://doi.org/10.1145/275107.275118>
 24. Shi, Y., Huang, Z., Feng, S., Zhong, H., Wang, W., Sun, Y.: Masked label prediction: Unified message passing model for semi-supervised classification. In: Zhou, Z. (ed.) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021. pp. 1548–1554. ijcai.org (2021). <https://doi.org/10.24963/IJCAI.2021/214>, <https://doi.org/10.24963/ijcai.2021/214>
 25. Shi, Z., Zheng, Z., Khan, S., Zhong, J., Li, M., Xu, Q.: Deepgate3: towards scalable circuit representation learning. *arXiv preprint arXiv:2407.11095* (2024)
 26. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: Verified propagation redundancy and compositional UNSAT checking in cakeml. *Int. J. Softw. Tools Technol. Transf.* **25**(2), 167–184 (2023)
 27. Temel, M., Slobodová, A., Hunt, W.A.: Automated and scalable verification of integer multipliers. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12224, pp.

- 485–507. Springer (2020). https://doi.org/10.1007/978-3-030-53288-8_23, https://doi.org/10.1007/978-3-030-53288-8_23
28. Ustun, E., Deng, C., Pal, D., Li, Z., Zhang, Z.: Accurate operation delay prediction for FPGA HLS using graph neural networks. In: IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020. pp. 87:1–87:9. IEEE (2020). <https://doi.org/10.1145/3400302.3415657>, <https://doi.org/10.1145/3400302.3415657>
 29. Wang, Z., Bai, C., He, Z., Zhang, G., Xu, Q., Ho, T.Y., Huang, Y., Yu, B.: Fgmn2: A powerful pre-training framework for learning the logic functionality of circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024)
 30. Wolf, C.: Yosys open synthesis suite. <https://yosyshq.net/yosys/>
 31. Wu, N., Li, Y., Hao, C., Dai, S., Yu, C., Xie, Y.: Gamora: Graph learning based symbolic reasoning for large-scale boolean networks. In: 2023 60th ACM/IEEE Design Automation Conference (DAC). pp. 1–6. IEEE (2023)
 32. Yu, C., Brown, W., Liu, D., Rossi, A., Ciesielski, M.: Formal verification of arithmetic circuits by function extraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **35**(12), 2131–2142 (2016)
 33. Yu, C., Ciesielski, M., Mishchenko, A.: Fast algebraic rewriting based on and-inverter graphs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **37**(9), 1907–1911 (2017)
 34. Yu, C., Ciesielski, M.J.: Automatic word-level abstraction of datapath. In: IEEE International Symposium on Circuits and Systems, ISCAS 2016, Montréal, QC, Canada, May 22–25, 2016. pp. 1718–1721. IEEE (2016). <https://doi.org/10.1109/ISCAS.2016.7538899>, <https://doi.org/10.1109/ISCAS.2016.7538899>
 35. Zhang, H., Jiang, J.R., Amarù, L.G., Mishchenko, A., Brayton, R.K.: Deep integration of circuit simulator and SAT solver. In: 58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5–9, 2021. pp. 877–882. IEEE (2021)

A Word-Level Arithmetic Block Identification via Cut Enumeration

To enable word-level feature encoding on an optimized AIG, we first perform reverse identification of small arithmetic primitives (XOR/MAJ, half adders, and full adders) using *cut enumeration*. We briefly recall the notions of k -feasible cuts and cut enumeration in the AIG setting. For readers interested in further algorithmic details, we refer to Section 3 (*Cut Computation*) of [6]; moreover, the ABC codebase provides additional practical examples of arithmetic reverse engineering built on cut enumeration (see the `accec` directory).

Cuts and k -feasible cuts in an AIG. Let G be an AIG and let v be a node in G . A *cut* c of v is a set of nodes in the transitive fan-in of v such that every path from any primary input to v passes through at least one node in c . A cut is *irredundant* if none of its proper subsets is also a cut of v . A cut c is a *k -feasible cut* if it is irredundant and $|c| \leq k$. *Cut enumeration* refers to enumerating all k -feasible cuts for each node, which provides a compact way to represent candidate local subcircuits rooted at v .

Truth-table extraction and Boolean-function matching. For each node v , we enumerate its k -feasible cuts and compute the truth table of the corresponding cut function. We maintain an offline precomputed database that stores the set of truth tables that are *NPN-equivalent* to target Boolean primitives, in particular XOR and MAJ functions (e.g., XOR2/XOR3 and MAJ2/MAJ3), where NPN-equivalence accounts for input negations, input permutations, and optional output negation. A cut is classified as an XOR (resp. MAJ) candidate if its truth table matches any entry in the corresponding NPN-class database.

Recovering half adders and full adders. After collecting all XOR and MAJ candidates, we identify half adders (HA) and full adders (FA) by pairing compatible candidates that share the same input nodes. Concretely, an HA is detected by a pair consisting of an XOR2 node (sum) and an MAJ2-equivalent node (carry) that share the same two inputs. An FA is detected by a pair consisting of an XOR3 node (sum) and an MAJ3 node (carry) that share the same three inputs. For each proposed (sum, carry) pair, we additionally validate that, after an appropriate NPN transformation, the pair implements the canonical HA/FA truth-table specification. Nodes covered by validated pairs are annotated as HA/FA blocks; remaining matched nodes are annotated as standalone XOR (or MAJ) primitives and used as word-level features in the subsequent GNN inference.

B Details of GNN Predictions

In this appendix, we would like to provide the details of GNN predictions given all available templates. The architecture is denoted by the following format “FirstStage_SecondStage_LastStage” representing the choices for PPG, PPA

and FSA, respectively. The multipliers are 64-bit wide optimized with dc2. As can be seen from Table 9, there is little challenge to predict PPG and PPA, while occasionally, FSA may be mispredicted.

Table 9: Top-3 predictions for dc2-optimized circuits

#	Tested dc2-opt circuit	rank-1 prediction	rank-2 prediction	rank-3 prediction
1	SP_4to2_BK	SP_4to2_BK	SP_4to2_LF	SP_4to2_JCA
2	SP_4to2_HCA	SP_4to2_HCA	SP_4to2_BK	SP_4to2_LF
3	SP_4to2_JCA	SP_4to2_LF	SP_4to2_JCA	SP_4to2_KS
4	SP_4to2_KS	SP_4to2_KS	SP_4to2_HCA	SP_4to2_LF
5	SP_4to2_LF	SP_4to2_LF	SP_4to2_JCA	SP_4to2_KS
6	SP_4to2_RC	SP_4to2_RC	SP_4to2_CK	SP_4to2_CL
7	SP_DT_BK	SP_DT_BK	SP_DT_LF	SP_DT_JCA
8	SP_DT_HCA	SP_DT_HCA	SP_DT_KS	SP_DT_LF
9	SP_DT_JCA	SP_DT_LF	SP_DT_JCA	SP_DT_HCA
10	SP_DT_KS	SP_DT_KS	SP_DT_HCA	SP_DT_LF
11	SP_DT_LF	SP_DT_LF	SP_DT_JCA	SP_DT_HCA
12	SP_DT_RC	SP_DT_RC	SP_DT_CK	SP_DT_CL
13	SP_WT_BK	SP_WT_BK	SP_WT_LF	SP_WT_JCA
14	SP_WT_HCA	SP_WT_HCA	SP_WT_BK	SP_WT_LF
15	SP_WT_JCA	SP_WT_LF	SP_WT_JCA	SP_WT_BK
16	SP_WT_KS	SP_WT_KS	SP_WT_HCA	SP_WT_RC
17	SP_WT_LF	SP_WT_LF	SP_WT_JCA	SP_WT_BK
18	SP_WT_RC	SP_WT_RC	SP_WT_CK	SP_WT_CL
19	BP_4to2_BK	BP_4to2_BK	BP_4to2_LF	BP_4to2_JCA
20	BP_4to2_HCA	BP_4to2_HCA	BP_4to2_KS	BP_4to2_BK
21	BP_4to2_JCA	BP_4to2_LF	BP_4to2_JCA	BP_4to2_KS
22	BP_4to2_KS	BP_4to2_KS	BP_4to2_HCA	BP_4to2_LF
23	BP_4to2_LF	BP_4to2_LF	BP_4to2_JCA	BP_4to2_KS
24	BP_4to2_RC	BP_4to2_RC	BP_4to2_CK	BP_4to2_CL
25	BP_DT_BK	BP_DT_BK	BP_DT_LF	BP_DT_JCA
26	BP_DT_HCA	BP_DT_HCA	BP_DT_BK	BP_DT_LF
27	BP_DT_JCA	BP_DT_LF	BP_DT_JCA	BP_DT_KS
28	BP_DT_KS	BP_DT_KS	BP_DT_HCA	BP_DT_LF

#	Tested dc2-opt circuit	rank-1 prediction	rank-2 prediction	rank-3 prediction
29	BP_DT_LF	BP_DT_LF	BP_DT_JCA	BP_DT_KS
30	BP_DT_RC	BP_DT_RC	BP_DT_CK	BP_DT_CL
31	BP_WT_BK	BP_WT_BK	BP_WT_LF	BP_WT_HCA
32	BP_WT_HCA	BP_WT_HCA	BP_WT_LF	BP_WT_BK
33	BP_WT_JCA	BP_WT_LF	BP_WT_JCA	BP_WT_BK
34	BP_WT_KS	BP_WT_KS	BP_WT_HCA	BP_WT_RC
35	BP_WT_LF	BP_WT_LF	BP_WT_JCA	BP_WT_BK
36	BP_WT_RC	BP_WT_RC	BP_WT_CK	BP_WT_CL
37	SP_AR_BK	SP_AR_SE	SP_AR_CL	SP_AR_HCA
38	SP_AR_CK	SP_AR_CK	SP_AR_RC	SP_AR_CL
39	SP_AR_CL	SP_AR_SE	SP_AR_CL	SP_AR_HCA
40	SP_AR_KS	SP_AR_KS	SP_AR_LF	SP_AR_RC
41	SP_AR_LF	SP_AR_SE	SP_AR_CL	SP_AR_HCA
42	SP_AR_RC	SP_AR_RC	SP_AR_CL	SP_AR_CK
43	SP_AR_SE	SP_AR_SE	SP_AR_CL	SP_AR_HCA
44	SP_CWT_BK	SP_CWT_BK	SP_CWT_LF	SP_CWT_JCA
45	SP_CWT_CK	SP_CWT_RC	SP_CWT_CK	SP_CWT_CL
46	SP_CWT_CL	SP_CWT_CL	SP_CWT_SE	SP_CWT_HCA
47	SP_CWT_KS	SP_CWT_KS	SP_CWT_HCA	SP_CWT_LF
48	SP_CWT_LF	SP_CWT_LF	SP_CWT_JCA	SP_CWT_BK
49	SP_CWT_RC	SP_CWT_RC	SP_CWT_CK	SP_CWT_CL
50	SP_CWT_SE	SP_CWT_SE	SP_CWT_CL	SP_CWT_HCA
51	SP_DT_BK	SP_DT_BK	SP_DT_SE	SP_DT_RC
52	SP_DT_CK	SP_DT_CK	SP_DT_RC	SP_DT_CL
53	SP_DT_CL	SP_DT_CL	SP_DT_LF	SP_DT_HCA
54	SP_DT_KS	SP_DT_KS	SP_DT_HCA	SP_DT_RC
55	SP_DT_LF	SP_DT_KS	SP_DT_LF	SP_DT_JCA
56	SP_DT_RC	SP_DT_RC	SP_DT_CK	SP_DT_CL
57	SP_DT_SE	SP_DT_SE	SP_DT_CL	SP_DT_HCA
58	SP_WT_BK	SP_WT_BK	SP_WT_LF	SP_WT_JCA
59	SP_WT_CK	SP_WT_CK	SP_WT_RC	SP_WT_CL
60	SP_WT_CL	SP_WT_CL	SP_WT_LF	SP_WT_HCA
61	SP_WT_KS	SP_WT_KS	SP_WT_HCA	SP_WT_RC
62	SP_WT_LF	SP_WT_LF	SP_WT_JCA	SP_WT_HCA
63	SP_WT_RC	SP_WT_RC	SP_WT_CK	SP_WT_CL
64	SP_WT_SE	SP_WT_SE	SP_WT_CL	SP_WT_HCA

C Detailed Equivalence Checking Time

This section presents a complete list of equivalence checking time, using 128-bit multipliers optimized by resyn3.

Table 10: Time and memory consumption of equivalence checking

#	Tested_resyn3_128_circuit	Time (seconds)	Memory (KB)	Method
1	SP_4to2_BK	61.60	78184	kissat
2	SP_4to2_HCA	81.04	87080	sat-sweeping
3	SP_4to2_JCA	125.33	81688	kissat
4	SP_4to2_KS	275.53	86572	kissat
5	SP_4to2_LF	126.82	86368	kissat
6	SP_4to2_RC	0.69	85356	fraig
7	SP_DT_BK	47.19	104264	kissat
8	SP_DT_HCA	81.94	105936	kissat
9	SP_DT_JCA	54.29	104828	kissat
10	SP_DT_KS	116.78	112904	kissat
11	SP_DT_LF	56.57	103688	kissat
12	SP_DT_RC	0.89	111632	fraig
13	SP_WT_BK	50.87	104216	kissat
14	SP_WT_HCA	5.42	67748	sat-sweeping
15	SP_WT_JCA	62.08	105424	kissat
16	SP_WT_KS	179.35	111812	kissat
17	SP_WT_LF	62.93	104300	kissat
18	SP_WT_RC	41.98	108100	kissat
19	BP_4to2_BK	83.30	129412	sat-sweeping
20	BP_4to2_HCA	271.10	111212	sat-sweeping
21	BP_4to2_JCA	925.40	100176	kissat
22	BP_4to2_KS	589.88	126496	sat-sweeping
23	BP_4to2_LF	964.83	102392	kissat
24	BP_4to2_RC	1.57	107072	fraig
25	BP_DT_BK	205.22	116008	sat-sweeping
26	BP_DT_HCA	374.82	109848	kissat
27	BP_DT_JCA	357.18	111816	kissat
28	BP_DT_KS	672.73	115396	kissat
29	BP_DT_LF	348.69	111860	kissat
30	BP_DT_RC	0.88	105508	fraig
31	BP_WT_BK	237.07	108100	kissat
32	BP_WT_HCA	382.67	108456	kissat
33	BP_WT_JCA	645.68	112884	kissat
34	BP_WT_KS	667.53	111092	kissat
35	BP_WT_LF	637.84	108664	kissat
36	BP_WT_RC	269.41	110884	sat-sweeping

#	Tested_resyn3_128_circuit	Time (seconds)	Memory (KB)	Method
37	SP_AR_BK	43.62	76176	sat-sweeping
38	SP_AR_CK	6.14	130784	fraig
39	SP_AR_CL	4947.87	276568	sat-sweeping
40	SP_AR_KS	1257.36	233016	sat-sweeping
41	SP_AR_LF	461.84	238684	sat-sweeping
42	SP_AR_RC	5.01	126128	fraig
43	SP_AR_SE	4.54	129400	fraig
44	SP_CWT_BK	115.41	114264	kissat
45	SP_CWT_CK	37.21	126924	sat-sweeping
46	SP_CWT_CL	3524.06	186608	kissat
47	SP_CWT_KS	240.08	88308	sat-sweeping
48	SP_CWT_LF	116.71	124204	kissat
49	SP_CWT_RC	27.63	92696	sat-sweeping
50	SP_CWT_SE	1.07	117224	fraig
51	SP_DT_BK	39.06	106960	kissat
52	SP_DT_CK	28.19	66628	sat-sweeping
53	SP_DT_CL	2450.09	149456	kissat
54	SP_DT_KS	122.68	108596	kissat
55	SP_DT_LF	71.81	107388	kissat
56	SP_DT_RC	1.20	113388	fraig
57	SP_DT_SE	0.76	81376	fraig
58	SP_WT_BK	4.71	68092	sat-sweeping
59	SP_WT_CK	59.40	108668	kissat
60	SP_WT_CL	1020.57	127684	kissat
61	SP_WT_KS	110.56	87728	sat-sweeping
62	SP_WT_LF	78.41	114068	kissat
63	SP_WT_RC	40.93	94736	sat-sweeping
64	SP_WT_SE	1.36	116064	fraig

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





EvolveGen: Algorithmic Level Hardware Model Checking Benchmark Generation through Reinforcement Learning

Guangyu Hu¹, Xiaofeng Zhou¹, Wei Zhang¹, and Hongce Zhang²

¹ The Hong Kong University of Science and Technology
{ghuae,xzhoubu}@connect.ust.hk, wei.zhang@ust.hk

² The Hong Kong University of Science and Technology (Guangzhou)
hongcezh@hkust-gz.edu.cn

Abstract. The advancement of hardware model checking is critically dependent on high-quality benchmarks. However, the community faces a significant benchmark gap: existing benchmark suites are limited in number, primarily provided at representations like BTOR2 without access to the originating register-transfer-level (RTL) designs, and skewed toward extreme difficulty levels, where instances are either trivial or intractable. These limitations not only impede the rigorous evaluation of new verification techniques but also encourage the overfitting of solver heuristics to a narrow set of problems.

To address this challenge, we introduce **EvolveGen**, a novel framework that generates hardware model checking benchmarks by combining reinforcement learning (RL) with high-level synthesis (HLS). Our approach operates at an algorithmic level of abstraction, where an RL agent learns to construct computation graphs. By compiling these graphs with different synthesis directives, we generate pairs of functionally equivalent but structurally distinct hardware designs, thereby inducing challenging model checking instances. The solver’s execution time serves as a reward signal, guiding the agent to autonomously discover and generate “small-but-hard” instances that target solver-specific weaknesses. Our experiments show that **EvolveGen** efficiently creates a diverse benchmark set in various standard formats (e.g., AIGER, BTOR2), effectively revealing performance bottlenecks in state-of-the-art model checkers.

1 Introduction

The progress of hardware model checking, a cornerstone of hardware formal verification [1], is fundamentally tied to the quality and diversity of available benchmarks. However, the community faces a chronic *shortage*: despite annual calls for benchmarks from venues like the hardware model checking competition (HWMCC) [2], obtaining authentic open-source hardware designs remains a major challenge. Consequently, a significant portion of widely-used suites originates

* Both authors contributed equally to this work.

from software or protocol verification, with only about 67% of recent HWMCC instances providing synthesizable register-transfer-level (RTL) descriptions [2]. This scarcity is compounded by a severe distribution problem. Existing suites are often polarized, filled with instances that are either trivially easy or extremely hard. Furthermore, as model checkers advance, previously challenging benchmarks may become obsolete, creating a persistent lack of the “solver-specific-challenging” instances crucial for robust evaluation. This environment fosters a damaging cycle of “over-training,” where solver heuristics are excessively tuned to a narrow, potentially biased, and sometimes even “badly encoded” set of problems. It is beneficial and essential to have a systematic approach to generate diverse, structurally complex, and algorithmically challenging hardware model checking benchmarks.

Related work. There have been several prior works that automatically generate circuits to test verification tools, such as **AIGFuzz** [3], **AIGen** [4], and **FuzzBtor2** [5]. Machine learning techniques, such as reinforcement learning (RL), have also been integrated, for example, in **BanditFuzz** [6]. While prior works generate syntactic mutants to better trigger tool bugs, they cannot generate benchmarks whose difficulty stems from complex semantic and structural properties, which is crucial to stress the performance of hardware model checkers.

Contributions. This paper introduces **EvolveGen**, a novel framework that leverages RL and high-level synthesis (HLS) to generate benchmarks from a high-level algorithmic abstraction. The RL agent learns to construct a computation graph representing an algorithm. This graph is then compiled into two functionally equivalent but distinct hardware designs using varied HLS directives. A reward signal, based on a model checker’s solving time, guides the agent to autonomously discover “small-but-hard” problems. These benchmarks can be generated in large scale, and they also expose structural bottlenecks that cause solver heuristics to fail. This is the first framework to generate hardware benchmarks at an algorithmic level. Our evaluation shows that these benchmarks reveal performance differences among the state-of-the-art model checkers.

Structure of the Paper. The remainder of this paper is structured as follows. Section 2 provides the necessary background. Section 3 presents our motivating example and key observations. Section 4 details the proposed **EvolveGen** framework. Section 5 presents the experimental results. Finally, Section 6 concludes the paper.

2 Preliminaries

Hardware Model Checking. Hardware model checking automatically verifies whether a hardware design satisfies a given set of properties [7]. The design is typically represented as a state transition system, while properties can be specified using a certain temporal logic [8]. A model checker explores the state space of the design to either prove that a property holds for all possible behaviors or provide a

counterexample trace that demonstrates a violation. The performance and capabilities of model checkers are typically evaluated using benchmark suites like the HWMCC benchmark set, making the quality and diversity of these benchmarks critical for advancing the field.

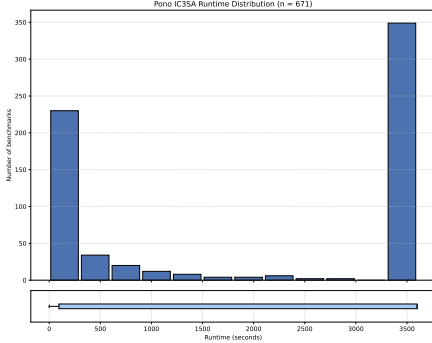
High-Level Synthesis. HLS is a design methodology that automates the creation of hardware from a high-level algorithmic description, typically written in languages like C, C++, or SystemC [9]. An HLS compiler analyzes the high-level code and synthesizes it into an RTL hardware implementation (e.g., in Verilog or VHDL). A key feature of HLS is the use of compiler directives, or *pragmas*, which allow designers to guide the synthesis process. For example, pragmas can specify how loops should be unrolled or pipelined and how arrays are mapped to memory. Two functionally identical C++ programs can be synthesized into vastly different hardware implementations by applying different pragmas, a characteristic that our framework leverages to create equivalence checking challenges.

Reinforcement Learning. Reinforcement Learning (RL) is a machine learning approach where an agent learns to make optimal decisions by interacting with an environment to maximize a cumulative reward signal [10]. RL problems are typically formalized as Markov Decision Processes (MDPs) [11] involving complex state transitions. The Multi-Armed Bandit (MAB) problem [12] represents a fundamental subclass of RL characterized by a single state and a finite set of available actions, $\mathcal{A}$. In the MAB setting, the agent repeatedly selects an action $a \in \mathcal{A}$ (often referred to as pulling an “arm”) and observes a stochastic reward drawn from an unknown underlying distribution. The objective is to maximize the total reward over time by balancing *exploration* (trying those less-frequently-tested arms) and *exploitation* (choosing the arm estimated to be the best).

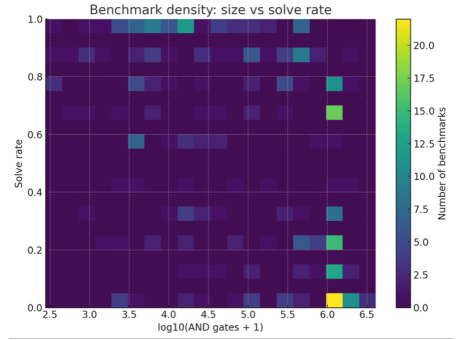
A powerful algorithm for solving MAB problems is *Thompson Sampling* [13], which employs a Bayesian approach for decision-making. In scenarios where rewards are binary (Bernoulli distributed), Thompson Sampling maintains a Beta distribution, $\text{Beta}(\alpha, \beta)$, for each arm to model the probability of receiving a reward. The shape parameters α and β correspond to the accumulated counts of successes (reward = 1) and failures (reward = 0), respectively. In each decision step, the agent samples a value from the Beta distribution of every arm and greedily selects the action corresponding to the highest sampled value. Upon observing the actual reward, the agent updates the α or β parameter of the chosen arm, thereby refining its belief model.

3 Motivation

Analysis of Existing Benchmarks. To advance hardware model checking, the community requires benchmarks that are not only challenging but also diverse in the nature of their difficulty. However, our analysis of the widely-used HWMCC’24 dataset reveals two critical limitations in the current benchmark landscape, as summarized in Fig. 1.



(a) Difficulty spectrum (solving time distribution by Pono)



(b) Benchmark density heatmap (size vs. time by all solvers that participated in HWMCC’24)

Fig. 1: Analysis of the HWMCC’24 benchmark suite. (a) The difficulty spectrum is highly polarized, with a scarcity of mid-difficulty instances (600s-3600s). Solving time is measured on Pono. (b) Benchmark difficulty is strongly correlated with circuit size, indicating scale-driven difficulty.

First, the difficulty spectrum is poorly distributed. As shown in Fig. 1a, the dataset is dominated by instances that are either trivially easy (solved in under 100 seconds) or intractably hard (unsolved after one hour). The crucial “mid-difficulty” range, which is most effective for distinguishing the performance of competitive solvers, is sparsely populated.

Second, benchmark difficulty is primarily driven by the scale of the problem, rather than the underlying structure. The heatmap in Fig. 1b reveals a notable correlation between circuit size (number of AIG gates) and the solving time. There is a severe lack of “small-but-hard” circuits, which are ideal for diagnosing solver behavior, and challenge a solver’s core *inductive reasoning* capabilities. When a solver fails on a small but hard instance, it points to a fundamental weakness in its reasoning logic, which might be obscured in a larger design. This observation highlights a clear need for benchmarks that are structurally complex and can rigorously test the inductive power of modern solvers.

A Motivating Example. Consider the simple C++ function shown in Fig. 2, which performs a multiply-accumulate operation within a loop. From this single high-level description, we can generate two functionally identical hardware implementations using HLS by applying different optimization directives (pragmas).

Version A is synthesized with the loop fully pipelined, resulting in a compact, sequential hardware architecture with a multiplier and an adder that are reused in each clock cycle. Version B, in contrast, is synthesized with the loop fully unrolled. The HLS compiler expands the loop into a large, parallel-dataflow architecture. Although both versions are algorithmically equivalent, their result-

ing hardware micro-architectures are drastically different at RTL. Verifying their equivalence often poses a challenge for formal verification tools.

Existing tools, such as AIGFuzz [3], AIGen [4], FuzzBtor2 [5] and Bandit-Fuzz [6] are unaware of the high-level algorithmic structures that are essential in the verification complexity. They often generate problems that are trivial to solve and therefore not suitable to assess the performance of the model checkers.

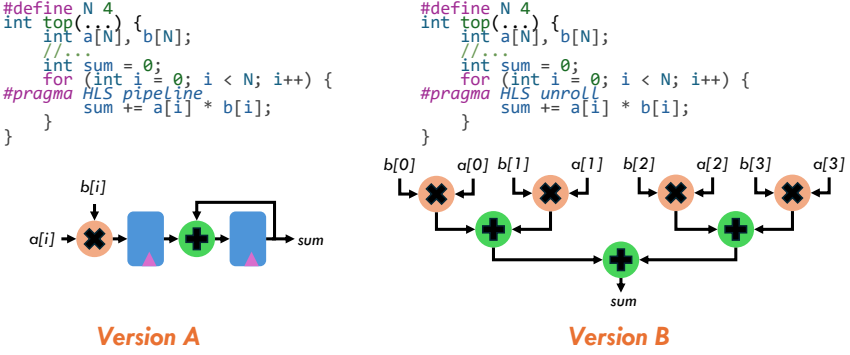


Fig. 2: Two functionally equivalent C++ implementations of a multiply-accumulate loop, differing only by HLS pragmas. Version A is pipelined, while Version B is fully unrolled, leading to vastly different hardware implementations.

4 The EvolveGen Framework

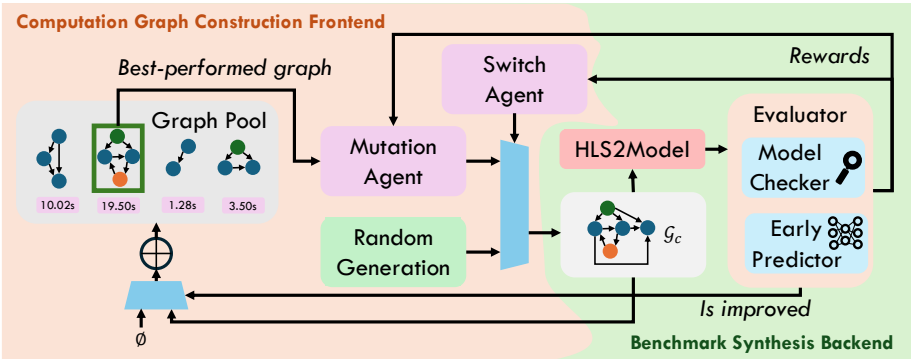


Fig. 3: The overall workflow of our proposed EvolveGen framework.

The overall workflow of **EvolveGen** is illustrated in Fig. 3. The framework is composed of two main components: a **Computation Graph Construction Frontend** and a **Benchmark Synthesis Backend**.

The frontend is responsible for generating a computation graph, a data structure used to represent the HLS design under verification in our **EvolveGen** framework. It also controls the high-level structure of the HLS-generated hardware. The backend takes the computation graph as input, and processes the computation graph with an **HLS2Model** tool. This **HLS2Model** tool converts a computation graph into a corresponding model checking problem. The backend also contains an evaluator to evaluate the performance of the generated benchmark model checking problem, and generate feedback to the frontend of **EvolveGen**.

In the remainder of this section, we will first introduce in detail the definition of the computation graph and how a selected computation graph is converted to a model checking problem in Section 4.1 and Section 4.2, respectively. Then, in Section 4.3, we will introduce the detailed RL-guided process in the frontend to generate the computation graph under evaluation. In this Section, we will also introduce the technique to generate the reward signal for the frontend computation graph generation part.

4.1 Computation Graph Representation

At the core of our framework is a directed graph representation of a program, which we call a computation graph. The nodes represent operations or control structures, and the edges represent data flow or membership of code blocks. We define four types of nodes, with examples in Fig. 4. During graph construction, the RL agent will attempt to add these types of nodes, representing corresponding graph transformation operations, which will be discussed in Section 4.3. The four types of nodes are:

- **OpNode**: represents a computation, such as arithmetic (ADD, MUL), bitwise (AND, XOR), or comparison (EQ, LT) operations. Each OpNode has attributes defining its data type (e.g., `ap_int<32>`, representing a 32-bit integer) and the operation type.
- **LoopNode**: represents a for-loop, with attributes for the start index, the ending index, and the step size. It also stores HLS pragma information, such as whether the loop is pipelined or unrolled.
- **BranchNode**: represents an if-then-else conditional branch. It is controlled by a predecessor OpNode whose result is treated as the branch condition.
- **DepNode**: is a special node, representing an inter-iteration data dependency within a loop (e.g., $a[i] = a[i - 1] + c$, where i is the loop variable). There is also such an example in the last graph in Fig. 4 where a depends on the value of b in the previous iteration. This is crucial for creating complex sequential behavior that is often challenging for model checkers.

As presented above, a node carries the associated attributes listed in Table 1. These attributes summarize the customizations that can be done to an HLS-generated design. They encompass most of the design options for fixed-point

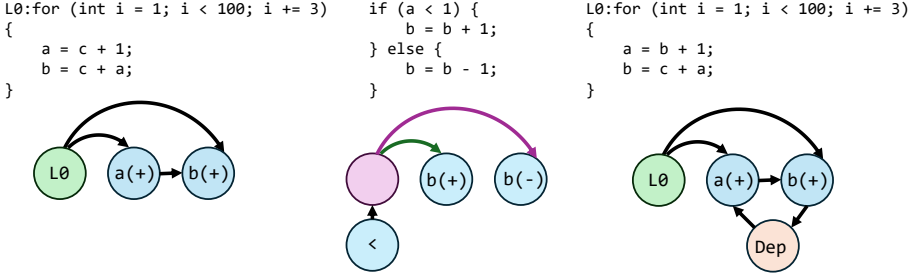


Fig. 4: Examples of different type of nodes in a computation graph with the corresponding code snippet, where c is a constant.

Table 1: Node types and their attributes.

Node type	Attributes
OpNode	operation type, data width, data type, rounding mode, saturation mode, and the integer value bits count
LoopNode	whether pipelined, whether flattened, loop variable information (including starting index, ending index, and the step size), unrolling mode (unroll factor, whether fully unrolled)
BranchNode	the branch condition
DepNode	N/A

datatypes and loops defined in the HLS tool, enabling the generated computation graphs to capture a rich variety of HLS design behaviors. It should be noted that in our design of the computation graph, *LoopNode* and *BranchNode* are also used to represent the corresponding code block. The successor nodes of these nodes are the contents of the code block.

4.2 Benchmark Synthesis from Computation Graphs

A computation graph can be transformed into a model checking problem using the *HLS2Model* process. Once the computation graph is finalized, *HLS2Model* constructs two functionally equivalent hardware designs, generated under distinct HLS optimization directives (pragmas). These differing settings result in hardware implementations that are structurally divergent but generate the identical computation result, enabling functional equivalence checking between them.

Note that the equivalence query in our generated problem is high-level and functional, representing a flexible notion of equivalence rather than the traditional combinational/sequential equivalence that requires either exactly the same combinational logic functions or cycle-by-cycle matching. As hardware verification tasks are mostly checking a certain kind of match between a specification and an implementation, which can often be formulated as a certain notion of

equivalence, the equivalence checking problem we constructed mimic this flexible notion of equivalence and is therefore representative in hardware verification.

As shown in Fig. 5, the `HLS2Model` can be split into four steps: C++ code generation, diversified HLS pragma insertion, miter generation, and model checking problem compilation.

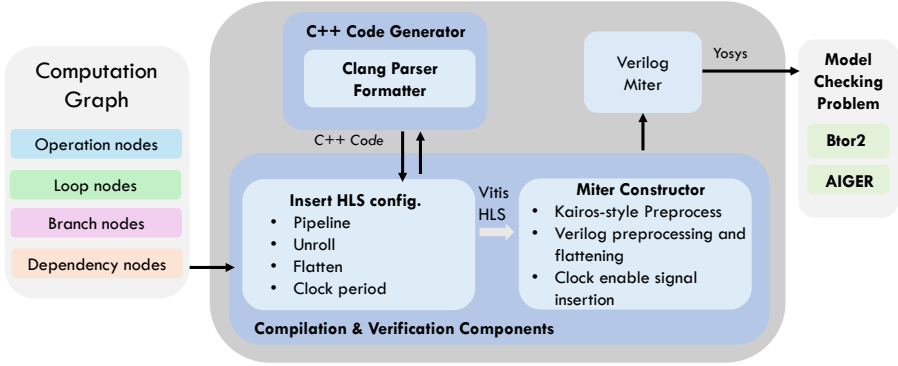


Fig. 5: The backend workflow: synthesizing computation graphs to hardware designs.

1. **C++ Code Generation:** The computation graph is traversed to generate a base C++ source file with the support of HLS-compatible arbitrary precision data types (e.g., `ap_int` and `ap_fixed`, representing the integer or fixed-point data types).
2. **HLS Pragma Insertion:** From the input computation graph, we create two functionally equivalent but structurally distinct variants by applying different sets of HLS pragmas. This is the primary mechanism for creating structural divergence for meaningful and challenging model checking problems. Our framework employs two distinct optimization strategies to generate two versions of hardware implementations for the computation graph:
 - **Basic:** This version is generated with minimal optimization pragmas. For instance, loops are synthesized using the HLS tool’s default sequential scheduling without pipelining, and a relaxed target clock period is specified.
 - **Optimized:** This version is aggressively optimized. For nested loops, inner loops are explicitly pipelined to maximize throughput, while outer loops are flattened if the nested loop is a perfect nested loop, i.e., there is no instruction between the execution of the outer and inner loop. Moreover, a stringent target clock period is enforced, encouraging the HLS compiler to insert more registers to break possible critical paths in the design. This guides the HLS tool towards a more parallel, high-performance architecture.

The application of these divergent strategies to the same source graph results in two distinct C++ files, which serve as the inputs for the subsequent equivalence checking problem.

3. **HLS Synthesis and Miter Generation:** Each C++ variant is synthesized into a Verilog module using a commercial HLS compiler [14]. The synthesized designs are then instantiated within a top-level *miter circuit* using the method in KAIROS [15], which is a functional equivalence checking framework that synchronizes the valid result production of HLS designs. The miter circuit connects their respective inputs to a common set of primary inputs and compares their outputs at the proper time. A dedicated single-bit output, `unsafe_signal` is added, which is asserted if and only if a functional discrepancy between the two designs is detected.
4. **Conversion to Verifier-Ready Format:** Finally, the Verilog miter circuit is processed through a logic synthesis flow [16] that converts the circuit into formats that can be accepted by the model checker, namely AIGER and BTOR2 format. This yields the final benchmark artifact.

4.3 Frontend RL-Guided Graph Generation

To discover computation graphs that produce challenging benchmarks, we approach the generation process using the reinforcement learning (RL) method. Our goal is to build RL agents to navigate the vast search space of possible graph structures and identify those that, when synthesized, result in computationally challenging verification problems for a targeted model checker. The entire process is guided by a hybrid evolutionary strategy, detailed in Algorithm 1. We will explain this method in two parts: (1) how different graph structures are constructed and (2) how rewards are computed to guide the RL agent to explore these different graph structures.

Multi-Agent Strategy for Graph Exploration. We adopt a multi-agent reinforcement learning (MARL) approach inspired by BanditFuzz [6], framing the generation process as a Multi-Armed Bandit (MAB) problem. There are in total two distinct agents that collaborate to balance exploration and exploitation: a high-level *Switch Agent* and a low-level *Mutation Agent*. This arrangement efficiently decomposes the learning task, allowing the Mutation Agent to focus purely on learning which structural operators yield hardness, while the Switch Agent independently manages population diversity, leading to faster convergence than a monolithic single-agent formulation and helps better escape local optima. Both agents employ the Thompson Sampling method [13]. The overall workflow, also depicted in Fig. 3, maintains a *graph pool* ($\mathcal{P}$) of promising graphs and iteratively seeks to improve it.

The algorithm takes two parameters as input: the maximum number of iterations and the initial pool size. The main loop (Line 4) is responsible for graph generation and it iterates until the pool reaches a target size. At each step, the *Switch Agent* first selects one of two strategies (Line 5):

Algorithm 1 The EvolveGen Algorithm**Require:** Maximum iterations M_{max} , Initial pool size N_{init} **Ensure:** A pool $\mathcal{P}$ of hard-to-solve benchmarks (graphs)

```

1: procedure EVOLVE( $M_{max}, N_{init}$ )
2:    $\mathcal{P} \leftarrow \text{INITIALIZEPOOL}(N_{init})$ 
3:    $\pi_{action}, \pi_{strategy} \leftarrow \text{InitializeAgents}()$ 
4:   while  $|\mathcal{P}| < M_{max}$  do
5:      $S \leftarrow \pi_{strategy}.\text{select\_action}()$   $\triangleright$  Choose mutate or generate new graph
6:     if  $S = \text{Mutate}$  and  $\mathcal{P} \neq \emptyset$  then
7:        $(\mathcal{G}_p, R_p) \leftarrow \text{SelectFromPool}(\mathcal{P})$ 
8:        $a \leftarrow \pi_{action}.\text{select\_action}()$ 
9:        $\mathcal{G}_c \leftarrow \text{Mutate}(\mathcal{G}_p, a)$ 
10:       $R_{new}, \text{status} \leftarrow \text{Evaluate}(\mathcal{G}_c)$ 
11:       $R_{base} \leftarrow R_p$ 
12:    else  $\triangleright S = \text{Generate}$ 
13:       $\mathcal{G}_c \leftarrow \text{GenerateFreshGraph}()$ 
14:       $R_{new}, \text{status} \leftarrow \text{Evaluate}(\mathcal{G}_c)$ 
15:       $R_{base} \leftarrow \text{AveragePoolPerformance}(\mathcal{P})$ 
16:    end if
17:    if status is success then
18:      improved  $\leftarrow R_{new} > R_{base}$ 
19:       $\pi_{strategy}.\text{reward}(\text{improved})$ 
20:      if  $S = \text{Mutate}$  then  $\pi_{action}.\text{reward}(\text{improved})$ 
21:      end if
22:      if improved then  $\mathcal{P} \leftarrow \mathcal{P} \cup \{(\mathcal{G}_c, R_{new}, \dots)\}$ 
23:      end if
24:    end if
25:  end while
26: end procedure

```

- **Mutate:** This exploitation-focused strategy can be chosen if the pool is not empty (Line 6). It selects a promising graph ($\mathcal{G}_p$) from the pool, prioritizing the graph with the highest predicted solving time (Line 7). The core mutation process is modeled as a Markov Decision Process (MDP), where the current graph is the state $s_t = \mathcal{G}_p$ of this MDP. The *Mutation Agent* then selects an action a_t (Line 8), which is a vector of graph transformation operators (e.g., adding a node from `<OpNode, BranchNode, LoopNode, DepNode>`) to apply to $\mathcal{G}_p$, yielding a new graph $\mathcal{G}_c$ (Line 9). A pair of rewards are computed (Line 10-11), where the new reward is evaluated following the method `Evaluate`, which will be discussed later, and the baseline reward for this action is the known parent’s reward, R_p . These two will be compared to help improve the policy of the agent.
- **Generate:** This exploration-focused strategy creates a completely new, randomly constructed graph from scratch (Line 13) using `GenerateFreshGraph`. This function takes a hyperparameter, which is the sequence length of ran-

dom construction actions. This hyperparameter is computed dynamically based on the average size of the current high-performing pool (capped at 40 actions) to strictly enforce the “small-but-hard” criterion. We will also evaluate the generated graph using the **Evaluate** function, which forms the new reward (Line 14), while the baseline reward for this choice is the average performance of the entire pool (Line 15), providing a general measure of quality.

After a new graph $\mathcal{G}_c$ is constructed either by Mutate or Generate, a reward R_{new} is determined, which is associated to the predicted solving time of the synthesized benchmark. If this reward is higher than the calculated baseline ($R_{new} > R_{base}$), the action is considered an “improvement” (Line 18). A positive feedback signal is then sent to update the agents’ policies (Lines 19-20), and the improved graph is added to the pool (Line 22). This reward-driven loop allows the agents to learn which strategies and mutations have historically yielded difficult benchmarks.

Early Prediction of Benchmark Difficulty. A key challenge in our RL loop is the high cost of obtaining an accurate reward signal, as a full model checking run is too slow for rapid training iterations. To address this, we developed a lightweight model to predict a benchmark’s solving time based on features extracted from a brief fixed-depth formal analysis.

This predictor is a gradient boosting regression model (XGBoost) [17] trained offline on a curated dataset. We collected data from the HWMCC’20 & ’24 competitions and constructed our dataset by selecting only those cases that could be solved within the first five recursion frames of PDR and completed in under 120 seconds. This selection criterion ensures the dataset is populated with benchmarks that yield informative short-run dynamic features relevant to our prediction task. The model learns a mapping from a vector of static and dynamic circuit features (see Table 2) to the final solving time. On our test set, the predictor achieved a high degree of accuracy, with R^2 scores of 0.60 for rIC3, 0.58 for IC3ref, and 0.46 for ABC PDR. This validates its use as a reliable reward proxy, enabling efficient RL-guided synthesis.

Table 2: Key features used by the model checking difficulty predictor model.

Category	Feature Class	Description and Rationale
Static	Size Metrics	Basic indicators of circuit scale (PI, flip-flop, AIG node counts).
	Structural Complexity	Measures of functional units (MUX, XOR, adder counts).
	Topological Depth	Captures the longest path and overall circuit topology (max/avg depth).
	Sequential Connectivity	Quantifies dependencies between state elements (flop fanout statistics).
Dynamic	PDR Proof Complexity	Internal statistics from a short PDR run (e.g., number of generated clauses, proof obligations, and SAT solver calls) to gauge initial proof effort.
	PDR Runtime Profile	Time distribution across PDR sub-procedures (e.g., total SAT solving, clause generalization, and clause pushing) to identify computational bottlenecks.
	Clause Generation Statistics	Metrics summarizing the distribution of new clauses per frame (e.g., mean, standard deviation, max), reflecting inductive difficulty.

The features consist of static features and the dynamic ones. **Static features** are derived directly from the AIG circuit’s structural properties without executing any verification tasks, providing a quick snapshot of its size and complexity. **Dynamic features**, conversely, are designed to capture the circuit’s effect on the formal proof search process, thereby revealing deeper properties related to its inductive complexity. To acquire these dynamic features efficiently, we perform a constrained fixed-depth (e.g., for 5 frames) run of the PDR algorithm. This short execution, while not intended to complete the proof, yields a rich set of predictive indicators. As detailed in Table 2, the dynamic features are grouped into three classes. As the PDR runtime profile is easy to understand, here we only explain the proof complexity and generation statistics in detail:

- **PDR Proof Complexity:** These features serve as direct measures of the proof work done by the PDR solver in its initial exploration. A high number of generated clauses suggests that the state space is complex and requires many lemmas to block bad states. Similarly, a large volume of proof obligations and SAT solver calls indicates that the solver is struggling to generalize lemmas and push them to later frames, a hallmark of a difficult inductive proof.
- **Clause Generation Statistics:** These features capture the core of inductive reasoning in PDR. Instead of merely counting new clauses, we analyze their *behavior* across frames. A key operation in PDR is “pushing” a learned clause (a lemma) to the highest possible frame, signifying its general inductive strength. When many new clauses are generated at a particular frame, it often indicates that existing lemmas are too specific and cannot be pushed, forcing the solver to re-discover invariants. We compute robust statistical metrics over the sequence of newly generated clauses per frame. A high mean or a large variance in this sequence signals significant inductive difficulty, as it implies the solver is struggling to find generalizable lemmas and is instead repeatedly generating new, localized ones that fail to propagate. This provides a direct measure of the proof’s inductive complexity.

The predicted solving time from this model serves as the reward signal R_{new} for our RL agents. This approach enables rapid iteration, allowing the agents to efficiently explore the vast search space of possible computation graphs without incurring the high cost of full model checking at each step.

5 Experimental Evaluation

Our experimental evaluation is designed to answer three research questions (Section 5.2–5.4) that assess the effectiveness of **EvolveGen** and the quality of the generated benchmarks.

- **RQ1:** Does **EvolveGen** generate challenging benchmarks efficiently?
- **RQ2:** Can **EvolveGen** generate high-quality benchmarks?
- **RQ3:** Can **EvolveGen** distinguish performance differences between two SOTA model checkers?

5.1 Experimental Setup

Target Model Checkers We use four state-of-the-art hardware model checkers: **rIC3** [18], **ABC-PDR** [19], **IC3Ref** [20] and **Pono** [21] as the model checkers guiding the benchmark generation. The predicted solving time of these model checkers on the generated model checking problem serves as the reward signal for our RL agent. We evaluate the generated benchmarks with these model checkers as well.

Baselines. We compare **EvolveGen** against several main baseline tools that can automatically produce model checking problems:

- **FuzzBtor** [5]: A state-of-the-art btor fuzzer that generate btor2 format problem for fuzzing model checker and SMT solvers.
- **AIGen** [4]: A tool that randomly generates boolean functions together with latches using uniform random sampling over the space of all boolean functions.
- **AIGFuzz** [3]: A fuzzer that randomly generates AIGs.

Environment. All experiments were conducted on a server equipped with an Intel 11th Gen CPU i5-11400 @ 4.400GHz and 128 GB of RAM, running Ubuntu 20.04. The timeout for each model checking run is set to 3600 seconds.

5.2 RQ1: Does EvolveGen generate challenging benchmarks efficiently?

The first research question discusses whether **EvolveGen** can quickly derive challenging benchmarks compared with existing methods. We run a 24-hour generation for each tool to find challenging cases. Fig. 6 illustrates the evolution of the maximum solving time of generated model checking benchmarks over time. During the 24-hour generation process, we find that the baseline tools struggle to produce model checking problems that exceed a 10-minute solving time. In contrast, **EvolveGen**, guided by a RL-based generation strategy, can quickly identify patterns associated with hard-to-solve cases for a specific model checker by leveraging the reward signal indicating the expected solving time. This enables it to generate difficult model checking problems early in the generation process. In Fig. 6b, although **FuzzBtor** is eventually able to generate a benchmark that takes more than 1 hour to solve on the **ABC-PDR** model checker, it requires more time to discover such a challenging case compared to **EvolveGen**.

5.3 RQ2: Can EvolveGen generate high-quality benchmarks?

High-quality benchmarks are defined as “small-but-hard”, referring to circuits with modest sizes that nonetheless require substantial solving time, as discussed in Section 3. The heatmap in Fig. 1b, which plots solving time against circuit size reveals that the existing benchmark suite lacks “small-but-hard” (high-quality)

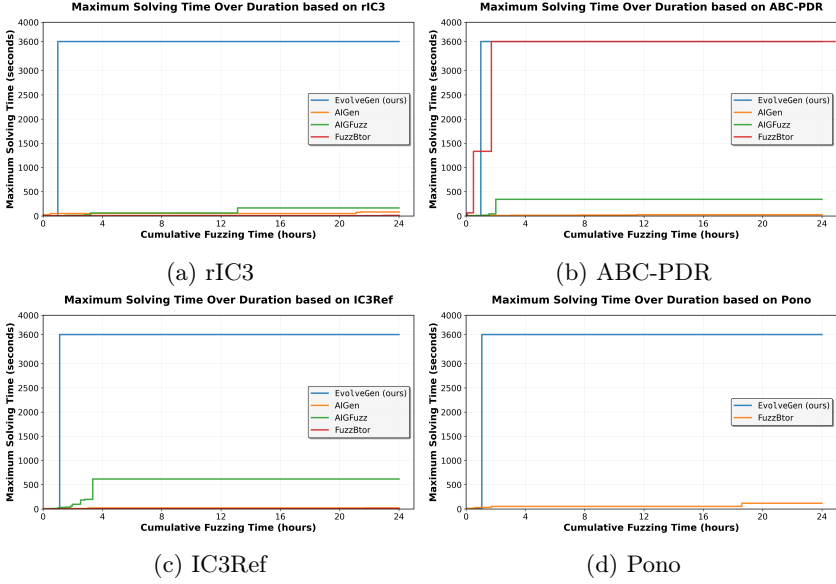


Fig. 6: Maximum solving time of derived model checking problems over time

cases. To assess benchmark quality from **EvolveGen**, we use the quality ratio (QR), defined in Eq. 1. Table 3 compares the QR values of existing benchmark generation tools and **EvolveGen**. It reports the average circuit size and solving time of the 10 most difficult cases produced during the 24-hour period.

From the results, we observe that AIGen, although capable of generating large AIGs, produces benchmarks that are relatively easy to solve. Across all model checkers, the hardest cases require less than two minutes to be solved on average. AIGFuzz shows similar behavior: it tends to generate large circuits, but the resulting model checking problems are generally not challenging. FuzzBtor exhibits this pattern only on IC3Ref, rIC3, and Pono. Although FuzzBtor can generate benchmarks that stress the ABC-PDR solver (shown in RQ1, Fig. 6b), the difficulty primarily arises from the excessive circuit size rather than intrinsic complexity of state space.

In contrast, our proposed method, **EvolveGen**, consistently produces model checking problems that are significantly harder to solve, while maintaining a relatively compact circuit size. This demonstrates its effectiveness in generating high-quality benchmarks that better satisfy the “small-but-hard” criterion. Specifically, we define the quality of a generated instance as:

$$QR = \frac{\text{Solving time of generated case}}{\text{AND gate count} + \text{Latch count}} \quad (1)$$

Fig. 7 provides the distribution of QR for the top 100 most challenging cases generated by baseline tools and all generated cases in our model checkers. **EvolveGen** outperforms baseline tools by at least an order of magnitude in

Table 3: Comparison of different tools across various model checkers (for each model checker and each solver, we perform 24-hour generation and calculate the average of 10 most challenging cases, and $\overline{QR}$ is calculated by the ratio of the average solving time and the average benchmark size, and **normalized**). **Timeouts are calculated as 3600s.**

Solver	Metric	AIGen	AIGFuzz	FuzzBtor	EvolveGen
ABC-PDR	Size	720179	26851	162548	15279
	Time	17.03	243.38	3600.00	3600.00
	$\overline{QR}$	1.00×10^{-4}	3.84×10^{-2}	9.36×10^{-2}	1.00
IC3Ref	Size	719888	29079	194439	16329
	Time	17.78	294.58	3.73	3600.00
	$\overline{QR}$	1.12×10^{-4}	4.59×10^{-2}	8.73×10^{-5}	1.00
rIC3	Size	719930	29633	237740	20217
	Time	69.75	58.99	5.95	3600.00
	$\overline{QR}$	5.44×10^{-4}	1.12×10^{-2}	1.40×10^{-4}	1.00
Pono	Size	—	—	224363	4052
	Time	—	—	64.41	3465.77
	$\overline{QR}$	—	—	3.36×10^{-4}	1.00

QR in most of the cases, indicating that our method can generate small but challenging model checking problems. Although there are a few simple cases from **EvolveGen** that do not have high quality, it is primarily due to the misprediction of the early predictor. Nevertheless, owing to the short solving time, such cases can be efficiently identified and filtered out through quick testing, and therefore have minimal impact on the overall quality of the generated benchmark suite.

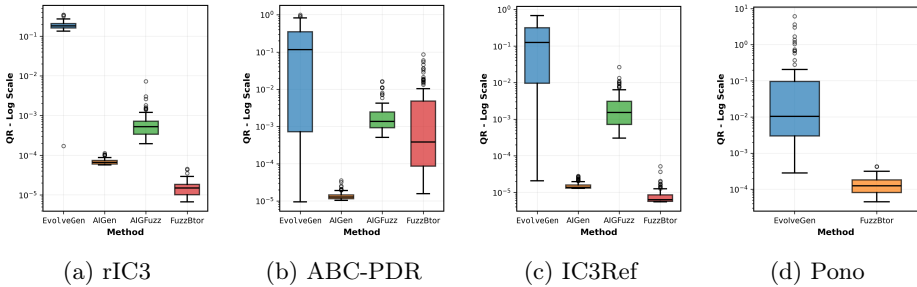


Fig. 7: Distribution of QR , for baseline tools, i.e., AIGen, AIGFuzz, and FuzzBtor, we pick the top 100 slowest generated benchmarks, and for **EvolveGen**, we use the whole generated benchmark suite.

5.4 RQ3: Can EvolveGen distinguish performance differences between two SOTA model checkers?

A key goal of a benchmark generator is to create instances that can clearly distinguish the performance of different solvers. We analyze whether the benchmarks generated by **EvolveGen** can reveal significant performance gaps between different state-of-the-art model checkers. Fig. 8 illustrates a performance comparison between two cutting-edge model checkers: **rIC3** (uses dynamic solving) and **Pono** (uses IC3 with syntax-guided abstraction) on our generated benchmark suite in previous experiments. From the comparison, we can draw two conclusions: (1) Our benchmark suite distinguishes the performance of different model checkers. As can be observed from the figure, the distribution of points is far from the line representing equal performance. (2) Owing to the multi-agent algorithms that dynamically mutate and produce the computation graph with the guidance of **Pono**, the distribution of the **Pono** solving time exhibits a more balanced distribution compared with the solving time of the original HWMCC benchmark suite mentioned in Fig. 1a. In the original HWMCC benchmark suite, the cases are either too simple or too hard to differentiate model checkers or provide meaningful evaluation for the performance of the model checker. According to the distribution on the right side of the figure, our **EvolveGen** can generate moderate-difficult cases for the target model checker. There is also an interesting observation: according to the result of HWMCC’24 [2], **rIC3** beat **Pono** in the word-level track, with our benchmark suite, **Pono** outperforms the originally state-of-the-art **rIC3**.

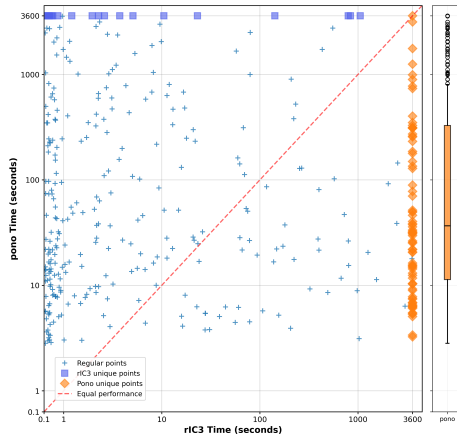


Fig. 8: Performance comparison between **rIC3** and **Pono** under the generated benchmark suite.

Discussion of Current Limitations. This work targets model checkers in the PDR family because currently PDR-based model checkers are among the most powerful ones in recent HWMCC. In theory, **EvoIveGen** can be extended to supporting algorithms like BMC and k-induction by extracting dynamic features from corresponding model checkers and re-training the predictor.

The model checking problems generated by **EvoIveGen** are functional equivalence checking problems between two hardware designs implementing the same high-level algorithm. This represents a common application scenario in hardware verification. The problem instances are not meant to completely replace existing HWMCC benchmarks but rather to offer a new perspective.

6 Conclusion

This paper addressed the critical challenge of benchmark scarcity in the hardware model checking domain. We introduced **EvoIveGen**, a novel framework that leverages reinforcement learning to guide the generation of benchmarks from a high-level algorithmic abstraction. By operating at the level of computation graphs and utilizing HLS to synthesize hardware, our approach systematically explores a rich space of structurally diverse and semantically complex designs.

Our experimental evaluation demonstrates that **EvoIveGen** is effective in generating high-quality benchmarks, controllable in its ability to produce circuits with desired characteristics, and capable of revealing performance bottlenecks in state-of-the-art model checkers. The results confirm that our RL-guided approach is significantly more efficient at discovering challenging instances than random fuzzing strategies.

For future work, we plan to extend the action space to include a wider range of algorithmic and hardware-specific constructs, such as memory access patterns and interface protocols. Another promising direction is to explore multi-objective reward functions that consider not only solving time but also specific structural features of the generated circuits, further enhancing the controllability and targeted nature of the benchmark generation process.

Acknowledgements. This work is supported by the Guangdong S&T Program (No. 2025A0505000022) and the Hong Kong Research Grants Council General Research Fund (GRF) (No. 16218324).

Code and Data Availability Statement. Our implementation, scripts, and documentation are available at: <https://github.com/xfzhou01/EvoIveGen>. Reproducing the full experimental results requires access to AMD Xilinx Vitis HLS, which is proprietary software and may require an appropriate license. Users can obtain Vitis HLS through the vendor’s official distribution and licensing channels.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Kern, C., Greenstreet, M.R.: Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 4(2), 123–193 (1999)
2. Biere, A., Froyleyks, N., Preiner, M.: Hardware model checking competition 2024 (2024), <https://hwmcc.github.io/2024/>
3. Zhang, C., Sun, M., Li, J., Su, T., Pu, G.: Feedback-guided circuit structure mutation for testing hardware model checkers. In: 2021 IEEE/ACM International Conference on Computer Aided Design (ICCAD). pp. 1–9 (2021). <https://doi.org/10.1109/ICCAD51958.2021.9643509>
4. Jacobs, S., Sakr, M.: AIGEN: Random generation of symbolic transition systems. In: *Computer Aided Verification (CAV 2021)*. Lecture Notes in Computer Science, vol. 12760, pp. 435–446. Springer International Publishing (2021). https://doi.org/10.1007/978-3-030-81688-9_20
5. Xiao, S., Zhang, C., Li, J., Pu, G.: FuzzBtor2: A random generator of word-level model checking problems in btor2 format. In: *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 79–87. Springer (2023)
6. Scott, J., Mora, F., Ganesh, V.: Banditfuzz: A reinforcement-learning based performance fuzzer for smt solvers. In: *International Workshop on Numerical Software Verification*. pp. 68–86. Springer (2020)
7. Clarke, E.M.: Model checking. In: *International conference on foundations of software technology and theoretical computer science*. pp. 54–56. Springer (1997)
8. Pnueli, A.: The temporal logic of programs. In: 18th annual symposium on foundations of computer science (SFCS 1977). pp. 46–57. IEEE (1977)
9. IEEE: IEEE standard for standard SystemC language reference manual. IEEE Computer Society (2012)
10. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, USA, 2nd edn. (2018)
11. Puterman, M.L.: Markov decision processes: discrete stochastic dynamic programming. John Wiley & Sons (2014)
12. Gittins, J., Glazebrook, K., Weber, R.: Multi-armed bandit allocation indices. John Wiley & Sons (2011)
13. Agrawal, S., Goyal, N.: Analysis of Thompson Sampling for the multi-armed bandit problem. In: *Conference on learning theory*. pp. 39–1. JMLR Workshop and Conference Proceedings (2012)
14. Xilinx: Vivado design suite user guide: High-level synthesis. UG902 (v2020.1) (May 2021)
15. Piccolboni, L., Di Guglielmo, G., Carloni, L.P.: KAIROS: Incremental verification in high-level synthesis through latency-insensitive design. In: 2019 Formal Methods in Computer-Aided Design (FMCAD). pp. 168–176 (2019). <https://doi.org/10.23919/FMCAD.2019.8894295>
16. Wolf, C., Glaser, J., Kepler, J.: Yosys: a free Verilog synthesis suite. In: *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*. vol. 97 (2013)
17. Chen, T., Guestrin, C.: XGBoost: A scalable tree boosting system. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. pp. 785–794. KDD '16, ACM, New York, NY, USA (2016). <https://doi.org/10.1145/2939672.2939785>
18. Su, Y., Yang, Q., Ci, Y., Bu, T., Huang, Z.: The rIC3 hardware model checker. arXiv preprint arXiv:2502.13605 (2025)

19. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: International Conference on Computer Aided Verification. pp. 24–40. Springer (2010)
20. Bradley, A.R.: SAT-based model checking without unrolling. In: Verification, Model Checking, and Abstract Interpretation. pp. 70–87. Springer (2011)
21. Mann, M., Irfan, A., Lonsing, F., Yang, Y., Zhang, H., Brown, K., Gupta, A., Barrett, C.: Pono: A flexible and extensible SMT-based model checker. In: Proceedings of the 33rd International Conference on Computer Aided Verification (CAV '21). Lecture Notes in Computer Science, vol. 12760, pp. 461–474. Springer International Publishing (2021)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Verifying First-Order Temporal Properties of Infinite-State Systems via Timers and Rankings

Raz Lotan^{1,2} , Neta Elad¹ , Oded Padon³ , and Sharon Shoham¹

¹ Tel Aviv University, Tel Aviv, Israel

² Certora, Tel Aviv, Israel

³ Weizmann Institute of Science, Rehovot, Israel

lotanraz@gmail.com

Abstract. We present a unified deductive verification framework for first-order temporal properties based on well-founded rankings, where verification conditions are discharged using SMT solvers. To that end, we introduce a novel reduction from verification of arbitrary temporal properties to verification of termination. Our reduction augments the system with prophecy timer variables that predict the number of steps along a trace until the next time certain temporal formulas, including the negated property, hold. In contrast to standard tableaux-based reductions, which reduce the problem to fair termination, our reduction does not introduce fairness assumptions. To verify termination of the augmented system, we follow the traditional approach of assigning each state a rank from a well-founded set and showing that the rank decreases in every transition. We leverage the recently proposed formalism of implicit rankings to express and automatically verify the decrease of rank using SMT solvers, even when the rank is not expressible in first-order logic. We extend implicit rankings from finite to infinite domains, enabling verification of more general systems and making them applicable to the augmented systems generated by our reduction, which allows us to exploit the decrease of timers in termination proofs. We evaluate our technique on a range of temporal verification tasks from previous works, giving simple, intuitive proofs for them within our framework.

1 Introduction

Several recent works [21, 22, 25, 26, 33] have dealt with verifying temporal properties of transition systems specified in first-order logic. Temporal properties of such systems are naturally specified in First-Order Linear Temporal Logic (FO-LTL). FO-LTL extends standard first-order logic with temporal operators that reason about the behavior of a transition system over time, and allows to specify both safety properties and liveness properties with various fairness assumptions.

Existing approaches for the verification of liveness properties of transition systems given in first-order logic can roughly be divided to those employing a liveness-to-safety reduction or those that use well-founded rankings. The liveness-to-safety reduction of [25] can target any FO-LTL property and is quite powerful

when augmented with temporal prophecy [15, 26]. However, verifying the safety of the resulting system is complicated since it requires reasoning about a monitored copy of the system that tracks repeated states modulo a dynamic finite abstraction. In contrast, well-founded rankings operate at the level of the original system and are therefore more intuitive to use, but current works in this setting [21, 22, 33] only target specific liveness properties and are limited in the kinds of ranking arguments they can capture.

Our goal is to design a verification framework for first-order transition systems that is both intuitive for users and applicable to any temporal property. To this end, we combine two key ideas. First, to obtain a unified approach, we base our framework on a novel reduction from the verification of FO-LTL properties to verification of termination. Our reduction is sound and complete and produces a transition system that is easy to reason about since it is an augmentation of the original transition system with prophecy variables that track violations of the property. Then, verifying termination of the resulting transition system is equivalent to showing that there is no violating trace of the original system. Second, for verifying termination, we adopt and extend the notion of implicit rankings from [21], allowing the user to encode a range of well-founded rankings that may involve the prophecy variables introduced by the reduction. Implicit rankings are provided by the user and verified automatically by an SMT solver.

Our reduction to termination takes a transition system T and a temporal property φ and reduces the verification of $T \models \varphi$ to verifying termination of an augmented transition system obtained by a synchronous composition of T with $T_{\neg\varphi}$, the *timer transition system* of $\neg\varphi$. The infinite traces of $T_{\neg\varphi}$ are exactly all the infinite traces that satisfy $\neg\varphi$. The timer transition system resembles standard tableaux constructions, except that it does not have fairness assumptions. As a result, we get a reduction to verification of termination instead of fair termination, which is easier to handle. To construct $T_{\neg\varphi}$ we introduce for each subformula ψ of $\neg\varphi$ a timer variable t_ψ that takes values in $\mathbb{N} \cup \{\infty\}$, and predicts the number of transitions until ψ is next satisfied, or ∞ if there is no such number. This behavior is enforced by the transitions of $T_{\neg\varphi}$. The initial states of $T_{\neg\varphi}$ are then defined as those where $t_{\neg\varphi} = 0$ which means that all of their outgoing infinite traces satisfy $\neg\varphi$. It follows that the composition $T \times T_{\neg\varphi}$ is an augmentation of T with timers that contains all infinite traces of T that satisfy $\neg\varphi$. Thus, $T \models \varphi$ holds if and only if the augmented transition system obtained by the composition has no infinite traces, that is, terminates.

To show termination of the transition systems produced by the reduction we employ the method of implicit rankings [21]. Implicit rankings are first-order formulas that capture the decrease of some ranking function between two states, without defining the function explicitly. Implicit rankings are useful for reasoning about ranking functions that are not easily expressible in first-order logic, such as functions that involve unbounded set cardinalities or summations. The constructors of implicit rankings defined in [21] provide an expressive language for well-founded rankings that can be reasoned about in first-order logic, using SMT solvers, but these are defined only for systems that operate over finite,

albeit unbounded, domains. In particular, they cannot leverage timers in rankings, even though their domain is well-founded. To mitigate this, we extend the formalism of implicit rankings to be able to soundly reason over systems with infinite domains. We do so by introducing an explicit soundness condition that allows us, for example, to require that some dynamically updated set is finite in all reachable states or that some relation is well-founded. We further generalize the constructors of [21] to this setting. In particular, this allows us to use the decrease of timers in the ranking used to prove termination of the augmented system produced by our reduction. Importantly, the use of timers in the termination proof relies on their intuitive meaning, and does not require understanding the construction of the augmented system.

We implement our reduction and the constructions of implicit rankings in a deductive verification tool that takes as input a transition system specification and a first-order temporal property, as well as an implicit ranking (given by a composition of constructors) and supporting invariants, which may refer to the timer variables. Our tool uses these ingredients to generate verification conditions that capture the decrease of rank encoded by the implicit rankings in every reachable transition of the system augmented with timers. It then verifies that the system satisfies the property by using an SMT solver to automatically discharge the verification conditions. We evaluate our tool by verifying a set of challenging examples from previous works which include liveness properties of distributed protocols and termination of programs. We give simple intuitive proofs for these examples within our framework.

Contributions.

- We introduce a reduction from the verification of arbitrary FO-LTL properties to verification of termination that augments the transition system with prophecy timer variables.
- We generalize the definition of implicit rankings and their constructors from recent work to settings with infinite domains.
- We combine the reduction to termination with implicit rankings to obtain a deductive verification framework where a user constructs an implicit ranking for the augmented system, and the decrease of rank in every step of the augmented system is verified automatically by an SMT solver.
- We implement and evaluate our approach on a set of transition systems and their properties from the literature.

The rest of the paper is organized as follows: Section 2 outlines a motivating example; Section 3 gives background on first-order logic and FO-LTL; Section 4 introduces the reduction from FO-LTL to termination; Section 5 presents the generalized definition of implicit rankings, the proof rule that uses them to prove termination and constructors for implicit rankings; Section 6 details our implementation and evaluation; Section 7 discusses related work and concludes the paper. We defer all proofs to the full version of the paper.

Full Version. The full version of the paper is available at [20].

2 Running Example

To illustrate our approach we consider the ticket protocol for ensuring mutual exclusion, a variant of Lamport’s bakery algorithm [18]. In the ticket protocol, access to the critical section is governed by natural-number tickets. The protocol maintains the next available ticket and the currently serviced ticket, both initialized to zero. Threads start at an idle state and may move to a waiting state by acquiring the next available ticket and in turn increasing it. Once in the waiting state, a thread must wait for its ticket to be the currently serviced ticket to enter the critical section. Immediately after a thread enters the critical section, it transitions back to the idle state while increasing the currently serviced ticket. The control-flow graph of each thread is presented in Fig. 1.

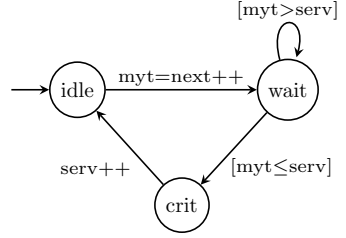


Fig. 1. Ticket Protocol

The temporal property we verify is that under fair scheduling, every thread that waits to enter the critical section, eventually does so. We formalize this property in FO-LTL by $\varphi = (\forall x \Box \Diamond \text{scheduled}(x)) \rightarrow (\forall x \Box (\text{waiting}(x) \rightarrow \Diamond \text{critical}(x)))$ where $\Box$ stands for “globally” and $\Diamond$ stands for “eventually”. To verify the property, we need to show that no infinite trace of the system satisfies the negation of φ . We do so by considering an augmented transition system that restricts the infinite traces of the original system to traces that satisfy the skolemized, negated property $\text{sk}(\neg\varphi) = \forall x. \Box \Diamond \text{scheduled}(x) \wedge \Diamond (\text{waiting}(x_0) \wedge \Box \neg \text{critical}(x_0))$. Verification is then reduced to showing that the augmented transition system has no infinite traces, i.e., that it terminates.

The augmented transition system is constructed automatically by composing the original one with a timer transition system for $\text{sk}(\neg\varphi)$. The general construction is given in Section 5.2. Here, we only illustrate the gist of the construction. The augmented transition system has a timer variable $t_{\text{starved}} \in \mathbb{N} \cup \{\infty\}$ and a timer function $t_{\text{sched}} : \text{Thread} \rightarrow \mathbb{N} \cup \{\infty\}$. The variable t_{starved} serves as a prophecy variable that predicts the number of steps until $\text{waiting}(x_0) \wedge \Box \neg \text{critical}(x_0)$ will next hold. When t_{starved} reaches 0 it means that x_0 is forever locked out of the critical section. The function $t_{\text{sched}}(x)$ predicts the number of steps until the thread x is next scheduled. The construction ensures that the timers are updated according to this meaning: if their value is positive, it decreases in every step; if it is infinite, it remains infinite; and if it is 0, it is ensured that the formula holds. Because we want to guarantee that $\text{sk}(\neg\varphi)$, the augmented transition system initializes t_{starved} to some natural number, it then decreases in every step of the system, which, due to the nature of the natural numbers, means it becomes 0 after finitely many steps. Additionally, to guarantee fair scheduling, for every thread x , the construction ensures that $t_{\text{sched}}(x)$ is equal to 0 infinitely often. It does so by initializing it to some natural number and resetting it to an arbitrary positive number whenever it reaches 0. Altogether this guarantees that *any* infinite trace of the augmented

system satisfies $\text{sk}(\neg\varphi)$ (without any fairness assumptions). It then remains to prove that the augmented system terminates.

In the remainder of the section we illustrate how timers are leveraged in the termination proof of the augmented system. We emphasize that to incorporate the timers in the proof, the user may rely on their intuitive meaning and need not worry about the construction of the augmented system. Our termination proof is based on a ranking function that maps system states to a rank from some well-founded set that decreases in every transition. However, instead of encoding the function explicitly, we use an extended notion of implicit rankings [21] which encode decrease in the rank between two states, and can be naturally defined using the constructors we provide (see Section 5.2). Due to space constraints, we only present the underlying ranking function. Encoding it as an implicit ranking using our constructors is rather straightforward, given our extension that allows to incorporate timers in implicit rankings despite their infinite domain.

We construct a lexicographic ranking function by considering the possible transitions of the (augmented) system. The first component of the rank is t_{starved} . This timer repeatedly decreases until it reaches 0, from which point on it remains 0, and x_0 is starved out of the critical section. The next components of the rank establish that the remainder of the trace from that point on is finite. To that end, we observe that since x_0 is in the waiting state indefinitely, its ticket value $\text{myt}(x_0)$ never changes and therefore the difference $\text{myt}(x_0) - \text{serv}$ never increases. When a thread leaves the critical state serv increases and the difference decreases. Therefore, we use the difference as the next component in the rank. Next, when no thread is in the critical section we wait for one to enter. Therefore, the third component in the rank captures whether there is a thread in the critical section. Finally, the scheduled thread may neither be in nor enter the critical section. To ensure decrease of rank in this case, we leverage the timers and introduce a fourth component of the rank that decreases according to $t_{\text{sched}}(x)$ for the thread x that holds the currently serviced ticket and is the next one to enter the critical section. When the resulting rank is encoded with an implicit ranking, the decrease of rank in every (reachable) step of the augmented transition system can be verified automatically by an SMT solver, ensuring termination.

3 Preliminaries

First-Order Logic. We use many-sorted first-order logic with equality. A signature Σ consists of a set of sorts, denoted $\text{sorts}(\Sigma)$, sorted relation, constant and function symbols. Terms are constant symbols, variables or function applications. We use boldface to denote a sequence of variables $\mathbf{x}$ or terms $\mathbf{t}$. Atomic formulas are $t_1 = t_2$ or relation applications $R(\mathbf{t})$. Non-atomic formulas are built using the connectives $\neg, \wedge, \vee, \rightarrow$ and sorted quantifiers $\forall, \exists$. For a formula α (or term), we write $\alpha(\mathbf{x})$ to denote that its free variables are contained in $\mathbf{x}$, and for a sequence of terms $\mathbf{t}$ we write $\alpha(\mathbf{t})$ for the simultaneous substitution $\alpha[\mathbf{x} \mapsto \mathbf{t}]$.

First-order formulas are evaluated over pairs of structures and assignments. A Σ -structure is a pair $s = (D, I)$ where D maps every sort σ to its domain $D(\sigma)$,

which is a non-empty set, and I is an interpretation function for all symbols in Σ . We denote by $\text{struct}(\Sigma, D)$ the set of all structures over D and by $\text{struct}(\Sigma)$ the class of all Σ -structures. A sorted assignment v from sorted variables $\mathbf{x}$ to D is a function that maps each variable x_i of sort σ in $\mathbf{x}$ to an element of $D(\sigma)$. We denote the set of all assignments from $\mathbf{x}$ to D by $\text{assign}(\mathbf{x}, D)$. For two assignments u, v over disjoint sequences of variables $\mathbf{x}, \mathbf{y}$, we denote by $u \cdot v$ the assignment to $\mathbf{x} \cdot \mathbf{y}$ defined by $u \cdot v(x_i) = u(x_i)$ and $u \cdot v(y_j) = v(y_j)$ for any i, j . For a formula α and a pair (s, v) of a structure s and an assignment v to the free variables of α we write $(s, v) \models \alpha$ to denote that (s, v) satisfies α .

For a signature Σ we denote by Σ' a disjoint copy of Σ defined by $\Sigma' = \{a' \mid a \in \Sigma\}$. For a sequence of variables $\mathbf{x} = (x_i)_{i=1}^m$, we denote by $\mathbf{x}'$ the sequence $(x'_i)_{i=1}^m$. For a formula α (or term t) over Σ , we denote by α' the formula over Σ' obtained by substituting each symbol $a \in \Sigma$ with $a' \in \Sigma'$. With abuse of notation we sometimes consider a Σ -structure $s = (D, I)$ as a Σ' -structure, or vice versa, by setting $I(a') = I(a)$. Similarly, we sometimes consider an assignment to $\mathbf{x}$ as an assignment to $\mathbf{x}'$, or vice versa, by setting $v(x'_i) = v(x_i)$. For a formula τ over $\Sigma \uplus \Sigma'$ with free variables $\mathbf{x}, \mathbf{x}'$ and structures s, s' over a shared domain, we use the notation $(s, v), (s', v') \models \tau$ to denote that τ is satisfied by the structure-assignment pair where the domain is the shared domain of s, s' and where $\Sigma, \mathbf{x}$ are interpreted according to (s, v) and $\Sigma', \mathbf{x}'$ are interpreted as in (s', v') . If τ is closed we omit the assignments and write $s, s' \models \tau$.

Specifying Transition Systems in First-Order Logic. A transition system over signature Σ , denoted $\mathcal{T}(\mathcal{S})$, is given by a pair of a first-order specification $\mathcal{T}$ and an intended semantics $\mathcal{S}$. The first-order specification is given by $\mathcal{T} = (\Gamma, \iota, \tau)$ where Γ is a closed formula over Σ that acts as a background theory, ι is a closed formula over Σ that defines the initial states of the system and τ is a closed formula over $\Sigma \uplus \Sigma'$ that defines the possible transitions of the system. The intended semantics $\mathcal{S}$ is a class of Σ -structures. States of $\mathcal{T}(\mathcal{S})$ are given by the Σ -structures in $\mathcal{S}$ that satisfy Γ . A trace of $\mathcal{T}(\mathcal{S})$ is given by a sequence of states $\pi = (s_i)_{i=0}^n$ where $n \in \mathbb{N} \cup \{\infty\}$ such that all states in the sequence have the same domain, $s_0 \models \iota$ and for every $i \leq n$ we have $s_i, s_{i+1} \models \tau$. A state s is reachable if there is a finite trace $\pi = (s_i)_{i=0}^n$ for $n \in \mathbb{N}$ such that $s_n = s$. A pair of states s, s' is a reachable transition if s is reachable and $s, s' \models \tau$. We say that $\mathcal{T}(\mathcal{S})$ *terminates* if it has no *infinite* traces.

First-Order Linear Temporal Logic. We use FO-LTL to specify temporal properties of transition systems. Given a first-order signature Σ , FO-LTL formulas are defined similarly to first-order formulas with the addition of temporal operators. For simplicity we only consider the temporal operators ‘globally’ $\Box$ and ‘eventually’ $\Diamond$, our approach can be easily extended to the operators ‘next’ $\bigcirc$ and ‘until’ $\mathcal{U}$. The semantics of an FO-LTL formula ψ is defined over a pair (π, v) , where π is an *infinite* sequence of Σ -structures $\pi = (s_i)_{i=0}^\infty$ over a shared domain, and v is an assignment to the free variables of ψ . For $\pi = (s_i)_{i=0}^\infty$ and $k \in \mathbb{N}$ we denote $\pi^k = (s_{i+k})_{i=0}^\infty$. The semantics of FO-LTL for ψ and (π, v) as above is such that if ψ is atomic we have $\pi, v \models \psi$ if and only if $s_0, v \models \psi$ in

the first-order semantics, $\pi, v \models \Box\psi$ if and only if $\pi^k, v \models \psi$ for every $k \in \mathbb{N}$, $\pi, v \models \Diamond\psi$ if and only if $\pi^k, v \models \psi$ for some $k \in \mathbb{N}$, and the first-order operators are interpreted in the usual way. A temporal property φ is given by a closed FO-LTL formula. A transition system $\mathcal{T}(\mathcal{S})$ satisfies a temporal property φ , denoted $\mathcal{T}(\mathcal{S}) \models \varphi$, if all of its infinite traces satisfy φ .

4 Timers

In this section we present the reduction from verification of FO-LTL properties to verification of termination. First, in Section 4.1, we present the timer transition system of a temporal property φ , and prove two characteristic facts about it. Then, in Section 4.2, we present the reduction, which uses the timer transition system of the negated temporal property to be verified.

4.1 The Timer Transition System

The timer transition system of a temporal property φ is a transition system constructed such that (i) all of its infinite traces satisfy φ and (ii) it contains all infinite traces that satisfy φ . We construct the timer transition system by considering a new sort, denoted σ_{time} , whose intended semantics are of the extended natural numbers $\mathbb{N} \cup \{\infty\}$, and adding for every subformula ψ a function symbol t_ψ called the *timer* of ψ .

Formally, for an FO-LTL property φ over signature Σ we denote by $A(\varphi)$ the set of formulas that contains all subformulas of φ , and for every subformula $\Box\psi$, additionally contains $\neg\psi$. The timer transition system $\mathcal{T}_\varphi(\mathcal{S}_{\text{time}})$ is defined over signature Σ_φ with sorts $\text{sorts}(\Sigma_\varphi) = \text{sorts}(\Sigma) \cup \{\sigma_{\text{time}}\}$ and signature $\Sigma_\varphi = \Sigma \cup \{t_\psi \mid \psi \in A(\varphi)\} \cup \{\leq, 0, \infty, -1\}$, where t_ψ for $\psi(\mathbf{x}) \in A(\varphi)$ is a function symbol from the sorts of $\mathbf{x}$ to σ_{time} ; $\leq$ is a binary relation symbol on σ_{time} ; 0 and ∞ are constant symbols of sort σ_{time} ; and -1 is a function symbol from σ_{time} to σ_{time} . The timer intended semantics $\mathcal{S}_{\text{time}}$ are given by the class of structures for Σ_φ where the domain of σ_{time} is $\mathbb{N} \cup \{\infty\}$, with the standard interpretations for $\leq, 0, \infty$ and -1 . The axioms and transition formula of $\mathcal{T}_\varphi$ are defined such that for any $\psi \in A(\varphi)$, $t_\psi(\mathbf{x})$ captures the number of transitions required until $\psi(\mathbf{x})$ is satisfied. Finally, the initial states formula ensures that $t_\varphi = 0$ holds in all initial states, which will imply that φ is satisfied in every trace.

Definition 1. Let φ be an FO-LTL property over signature Σ , and define $A(\varphi)$, Σ_φ and $\mathcal{S}_{\text{time}}$ as above. The timer transition system of φ is a transition system $\mathcal{T}_\varphi(\mathcal{S}_{\text{time}})$ over Σ_φ , where $\mathcal{T}_\varphi = (\Gamma_\varphi, \iota_\varphi, \tau_\varphi)$ is defined by:

$$\begin{aligned} \Gamma_\varphi &= \bigwedge_{\psi(\mathbf{x}) \in A(\varphi)} \forall \mathbf{x} ((t_\psi(\mathbf{x}) = 0) \leftrightarrow \zeta_\psi(\mathbf{x})) & \iota_\varphi &= (t_\varphi = 0) \\ \tau_\varphi &= \bigwedge_{\psi(\mathbf{x}) \in A(\varphi)} \forall \mathbf{x} \left(\begin{aligned} &(0 < t_\psi(\mathbf{x}) < \infty \rightarrow t'_\psi(\mathbf{x}) = t_\psi(\mathbf{x}) - 1) \wedge \\ &(t_\psi(\mathbf{x}) = \infty \rightarrow t'_\psi(\mathbf{x}) = \infty) \wedge \\ &(t_\psi(\mathbf{x}) = 0) \leftrightarrow \eta_\psi(\mathbf{x}) \end{aligned} \right) \end{aligned}$$

Table 1. Definitions of ζ_ψ, η_ψ for an FO-LTL formula $\psi(\mathbf{x})$

$\psi(\mathbf{x})$	ζ_ψ	η_ψ
$R(\mathbf{x})$	$R(\mathbf{x})$	
$\neg\rho$	$t_\rho(\mathbf{x}) \neq 0$	
$\rho \circ \chi$ for $\circ \in \{\wedge, \vee, \rightarrow\}$	$(t_\rho(\mathbf{x}) = 0) \circ (t_\chi(\mathbf{x}) = 0)$	
$Qy \rho$ for $Q \in \{\forall, \exists\}$	$Qy (t_\rho(\mathbf{x} \cdot y) = 0)$	
$\Diamond\rho$	$t_\rho(\mathbf{x}) < \infty$	$t_\rho(\mathbf{x}) = 0 \vee t'_{\Diamond\rho}(\mathbf{x}) = 0$
$\Box\rho$	$t_{\neg\rho}(\mathbf{x}) = \infty$	$t_\rho(\mathbf{x}) = 0 \wedge t'_{\Box\rho}(\mathbf{x}) = 0$

where ζ_ψ and η_ψ are defined according to Table 1. If η_ψ is undefined, the conjunct that refers to it is omitted from τ_φ .

Example 1. Consider $\varphi = \forall x. \Box \Diamond \text{sched}(x)$. The timers in $\mathcal{T}_\varphi$ are $t_{\forall x. \Box \Diamond \text{sched}(x)}$, $t_{\Box \Diamond \text{sched}(x)}(x)$, $t_{\neg \Diamond \text{sched}(x)}(x)$, $t_{\Diamond \text{sched}(x)}(x)$, $t_{\text{sched}(x)}(x)$. In initial states of $\mathcal{T}_\varphi$ we have $t_\varphi = 0$, and so from Γ_φ we have $\forall x. t_{\Box \Diamond \text{sched}(x)}(x) = 0$ in all initial states. From τ_φ it follows that $\forall x. t_{\Diamond \text{sched}(x)}(x) = 0$ in all reachable states. From the axiom generated for $\Diamond$, for any thread x , $t_{\text{sched}(x)}(x) < \infty$ in any reachable state. If $t_{\text{sched}(x)}(x)$, from decrease of timers in τ_φ it follows that this timer decreases to 0 in finitely many steps. Thus, for any thread x we have infinitely often $t_{\text{sched}(x)}(x) = 0$ in any trace, and so fair scheduling is guaranteed. That is, every infinite trace of $\mathcal{T}_\varphi$ satisfies φ .

Remark 1. The construction of the timer transition system resembles tableaux constructions [26, 32]. Instead of using sets of temporal formulas as states, we augment states with timers. In the tableau construction, fairness constraints are used to filter out undesirable traces, such as infinite traces where all states are marked $\Diamond p$ on the basis of some future satisfaction of p , which never arrives. Instead, in our construction, we rely on the intended semantics of timers to prevent such cases. Since such traces induce infinitely decreasing timer values, the well-foundedness of the natural numbers ensures that they do not exist.

Our first lemma states that for any $\psi \in A(\varphi)$ timers evaluated to 0 capture the satisfaction of the corresponding temporal formula. In ι_φ we require that $t_\varphi = 0$, so it follows that every infinite trace of $\mathcal{T}_\varphi(\mathcal{S}_{\text{time}})$ satisfies φ .

Lemma 1. *Let $\hat{\pi} = (\hat{s}_i)_{i=0}^\infty$ be a trace of $\mathcal{T}_\varphi(\mathcal{S}_{\text{time}})$ with domain D , $\psi(\mathbf{x}) \in A(\varphi)$, $v \in \text{assign}(\mathbf{x}, D)$ and $i \in \mathbb{N}$. Then $\hat{s}_i, v \models (t_\psi(\mathbf{x}) = 0)$ if and only if $\hat{\pi}^i, v \models \psi(\mathbf{x})$.*

The following lemma states that any sequence of states that satisfies φ can be augmented to produce a trace of $\mathcal{T}_\varphi(\mathcal{S}_{\text{time}})$. This essentially means that $\mathcal{T}_\varphi(\mathcal{S}_{\text{time}})$ exhibits all the ways in which φ can be satisfied. The proof of the lemma works by showing that we can always augment traces with the “natural definition” of timers, which is to evaluate $t_\psi(\mathbf{x})$ to the minimal number of transitions that should be taken until the suffix of the sequence satisfies ψ , or ∞ if none exists.

Lemma 2. *Given a sequence of Σ -structures $\pi = (s_i)_{i=0}^\infty$ over a shared domain such that $\pi \models \varphi$, there exists a trace $\hat{\pi} = (\hat{s}_i)_{i=0}^\infty$ of $\mathcal{T}_\varphi(\mathcal{S}_{\text{time}})$ such that $\hat{\pi}|_\Sigma = \pi$.*

4.2 The Timer Reduction

The reduction from satisfaction of FO-LTL properties to termination essentially composes $\mathcal{T}(\mathcal{S})$ with the timer transition system of the negated property $\mathcal{T}_{\neg\varphi}(\mathcal{S}_{\text{time}})$. Formally, for two transition systems over signatures Σ_1, Σ_2 with specifications $\mathcal{T}_1 = (\Gamma_1, \iota_1, \tau_1)$, $\mathcal{T}_2 = (\Gamma_2, \iota_2, \tau_2)$ and intended semantics $\mathcal{S}_1, \mathcal{S}_2$ respectively, we define their *product transition system* over $\Sigma_1 \cup \Sigma_2$ where $\Sigma_1 \cup \Sigma_2$ is not necessarily a disjoint union. The product transition system is denoted $(\mathcal{T}_1 \times \mathcal{T}_2)(\mathcal{S}_1 \times \mathcal{S}_2)$ and defined such that $\mathcal{T}_1 \times \mathcal{T}_2 = (\Gamma_1 \wedge \Gamma_2, \iota_1 \wedge \iota_2, \tau_1 \wedge \tau_2)$, and $\mathcal{S}_1 \times \mathcal{S}_2$ contains any $(\Sigma_1 \cup \Sigma_2)$ -structure s such that $s|_{\Sigma_1} \in \mathcal{S}_1$ and $s|_{\Sigma_2} \in \mathcal{S}_2$. In our case, $\mathcal{T}(\mathcal{S})$ and $\mathcal{T}_{\neg\varphi}(\mathcal{S}_{\text{time}})$ are defined over signatures Σ, Σ_φ respectively such that $\Sigma \subseteq \Sigma_\varphi$. Hence, the product transition system is defined over Σ_φ and can be understood as an augmentation of $\mathcal{T}(\mathcal{S})$ with the timer variables and semantics of $\mathcal{T}_{\neg\varphi}(\mathcal{S}_{\text{time}})$. The product system $\mathcal{T}(\mathcal{S}) \times \mathcal{T}_{\neg\varphi}(\mathcal{S}_{\text{time}})$ contains those traces of $\mathcal{T}(\mathcal{S})$ that satisfy $\neg\varphi$, it follows that $\mathcal{T}(\mathcal{S})$ satisfies the property if and only if the product system terminates. The following theorem summarizes the soundness and completeness of the reduction to termination.

Theorem 1. *For a transition system $\mathcal{T}(\mathcal{S})$ and an FO-LTL property φ we have $\mathcal{T}(\mathcal{S}) \models \varphi$ if and only if $\mathcal{T}(\mathcal{S}) \times \mathcal{T}_{\neg\varphi}(\mathcal{S}_{\text{time}})$ terminates.*

5 Rankings

In this section we discuss how we extend and use the framework of implicit rankings from [21] to prove termination of a transition system. Section 5.1 gives the definition of implicit rankings, generalized such that they are parameterized by an explicit soundness condition on reachable states of the system, and shows how implicit rankings are used in termination proofs. Section 5.2 introduces recursive constructions of implicit rankings based on the constructors of [21], extended to produce soundness conditions, and demonstrates how timers are leveraged in the construction of implicit rankings. In Section 5.3 we show how we discharge the proof obligations for a termination proof based on implicit rankings using validity checks in first-order logic.

5.1 Verifying Termination with Implicit Rankings

We first introduce our extended definition of *closed* implicit rankings, which are those used for proofs. We then give the definition of implicit rankings, that are not necessarily closed, which can be used for recursive constructions. For the remainder of the section, we fix a first-order signature Σ . To simplify the notation we assume that Σ is single-sorted. The generalization to many-sorted signatures, which we use in practice, is straightforward. A closed implicit ranking is a closed

formula $\tau_{>}$ that encodes the decrease of some implicit ranking function w.r.t. some implicit order. The formula $\tau_{>}$ is defined over a double signature $\Sigma \uplus \Sigma'$, where the rank decreases between the pre-state, captured by Σ , and the post-state, captured by Σ' . In [21], the implicit order is required to be well-founded. Instead, we relax the requirement and extend the notion of implicit rankings to include a soundness condition $\mathcal{C}$ on states, such that well-foundedness of the implicit order need only be guaranteed for the image of the ranking function on states that satisfy $\mathcal{C}$. We use the soundness condition to specify requirements that are necessary to guarantee well-foundedness of the implicit order but are not directly definable in first-order logic. The soundness conditions we use include the requirement of finiteness of some set of elements and the requirement of well-foundedness of some state relation.

Definition 2 (Closed Implicit Ranking). *Given a closed formula $\tau_{>}$ over $\Sigma \uplus \Sigma'$ and a class of Σ -structures $\mathcal{C}$, we say that $\tau_{>}$ is a closed implicit ranking with soundness condition $\mathcal{C}$ if for any domain D , there exist a partially ordered set $(A, <)$ and a function $f: \text{struct}(\Sigma, D) \rightarrow A$, such that the following hold:*

1. *for every $s, s' \in \text{struct}(\Sigma, D)$ we have $s, s' \models \tau_{>} \implies f(s) > f(s')$*
2. *$<$ restricted to $\{f(s) \mid s \in \mathcal{C}\}$ is well-founded.*

We call $(A, <)$ a ranking range for D and f a ranking function for D .

We note that for using implicit rankings in termination proofs it suffices to require item 1 (or even that f is defined) only for structures in $\mathcal{C}$, but this does not turn out to be helpful for constructing useful implicit rankings.

We use a closed implicit ranking $\tau_{>}$ with soundness condition $\mathcal{C}$ to show that a transition system $\mathcal{T}(\mathcal{S})$ terminates by showing that all reachable states of $\mathcal{T}(\mathcal{S})$ adhere to the soundness condition $\mathcal{C}$ and that $\tau_{>}$ is satisfied by all reachable transitions. Termination follows since if the system had an infinite trace, that would imply an infinitely decreasing sequence in a well-founded order. We formalize this in the following theorem:

Theorem 2. *If every reachable state s of $\mathcal{T}(\mathcal{S})$ satisfies $s \in \mathcal{C}$ and every reachable transition (s, s') of $\mathcal{T}(\mathcal{S})$ satisfies $s, s' \models \tau_{>}$ then $\mathcal{T}(\mathcal{S})$ terminates.*

In Section 5.3 we discuss how we verify the requirements of Theorem 2 in practice, before that, we present constructions of implicit rankings. To that end, we first generalize the definition. Similarly to [21], we observe that we can construct implicit rankings by identifying useful primitives and composing them. To allow such recursive constructions which use implicit rankings as building blocks for the construction of complex implicit rankings, [21] generalizes the definition of closed implicit rankings in two ways. First, implicit rankings may have free variables, called *parameters*, in which case the underlying (implicit) ranking function for each domain is defined over pairs of structures and assignments to the free variables. Parameters are useful for constructing implicit rankings that capture aggregations over the domain. Second, implicit rankings in [21] additionally include formulas $\tau_{\geq}$ that approximate the weak partial order $\leq$ over the ranking

range. This allows recursive constructions to rely on $\leq$ as well. We add a third generalization, which is a formula $\kappa_{\min}$ that approximates whether an element is of minimal rank. This generalization is used to define the soundness conditions of implicit rankings constructed via aggregations over infinite domains.

Definition 3 (Implicit Ranking). *An implicit ranking with parameters $\mathbf{x}$ and soundness condition $\mathcal{C}$ is a triple $\text{Rk} = (\tau_>, \tau_\geq, \kappa_{\min})$ where $\tau_>, \tau_\geq$ are formulas over $\Sigma \uplus \Sigma'$ with free variables $\mathbf{x}, \mathbf{x}'$, $\kappa_{\min}$ is a formula over Σ with free variables $\mathbf{x}$ and $\mathcal{C}$ is a class of structures over Σ , such that for any domain D , there exist a partially ordered set $(A, <)$ and a function $f: \text{struct}(\Sigma, D) \times \text{assign}(\mathbf{x}, D) \rightarrow A$, such that the following hold:*

1. $(s, v), (s', v') \models \tau_>(\mathbf{x}, \mathbf{x}') \implies f(s, v) > f(s', v')$
2. $(s, v), (s', v') \models \tau_\geq(\mathbf{x}, \mathbf{x}') \implies f(s, v) \geq f(s', v')$
3. $(s, v) \models \kappa_{\min} \implies f(s, v)$ is minimal in $(A, <)$
4. $<$ restricted to $\{f(s, v) \mid s \in \mathcal{C}, v \in \text{assign}(\mathbf{x}, D)\}$ is well-founded.

where conditions 1,2,3 range over any $(s, v), (s', v') \in \text{struct}(\Sigma, D) \times \text{assign}(\mathbf{x}, D)$. We call $(A, <)$ a ranking range for D and f a ranking function for D .

As with closed implicit rankings, we can weaken the requirements by only requiring conditions 1,2,3 for $s, s' \in \mathcal{C}$, but this does not turn out to be useful.

5.2 Constructors of Implicit Rankings

We revisit the constructors of [21]. For each, we additionally specify the definition of $\mathcal{C}$ and $\kappa_{\min}$ for the resulting implicit ranking. Moreover, we leverage the soundness condition for implicit rankings as defined in this work to generalize the finite-domain constructors of [21], which are only sound for sorts with finite domains, to allow weaker finiteness assumptions. Due to space constraints, we present only a subset of the constructors that are used to demonstrate the use of timers in ranking and those with non-trivial soundness conditions, all constructors appear in [20].

Theorem 3. *All the constructors presented are sound, in the sense that whenever their arguments satisfy their assumptions, they output an implicit ranking with suitable soundness conditions.*

The first constructor we present relies on an existing order over domain elements in the system itself to define an implicit ranking whose underlying ranking function decreases with respect to this order. The order is specified by a relation symbol $\ell(y_1, y_2)$, which is required to satisfy the formula $\text{order}(\ell)$ that specifies that ℓ is immutable and defines a strict partial order. The soundness condition of this implicit ranking is that the order defined by ℓ is well-founded. We require well-foundedness as a soundness condition as it is not possible to specify in first-order logic. While this may seem like deferring the problem rather than solving it, we are often able to leverage the intended semantics of the transition system, for example the intended semantics of timers $\mathcal{S}_{\text{time}}$, to ensure that the soundness condition holds without explicitly encoding it (see Section 5.3).

Constructor 1. *The position constructor takes a term t over Σ with free variables $\mathbf{x}$ and a binary relation symbol $\ell \in \Sigma$, returns an implicit ranking $\text{Pos}(t, \ell) = (\tau_{>}, \tau_{\geq}, \kappa_{\min})$ with parameters $\mathbf{x}$ and soundness condition $\mathcal{C}$ defined by:*

$$\begin{aligned}\tau_{>}(\mathbf{x}, \mathbf{x}') &= \text{order}(\ell) \wedge \ell(t'(\mathbf{x}'), t(\mathbf{x})) \\ \tau_{\geq}(\mathbf{x}, \mathbf{x}') &= \text{order}(\ell) \wedge (\ell(t'(\mathbf{x}'), t(\mathbf{x})) \vee t'(\mathbf{x}') = t(\mathbf{x})) \\ \kappa_{\min}(\mathbf{x}) &= \forall y. \neg \ell(y, t(\mathbf{x})) \quad \mathcal{C} = \{s \in \text{struct}(\Sigma) \mid \ell \text{ is well-founded in } s\}\end{aligned}$$

Example 2. Consider the ticket protocol of Section 2, after having augmented the system with timers. We use the timers in implicit rankings with the position constructor as follows. The relation ℓ is the order relation $<$ on σ_{time} . Taking $t = t_{\text{waiting}(x_0)} \wedge \Box \neg \text{critical}(x_0)$, we get the closed implicit ranking capturing the decrease of t_{starved} . Taking $t(x) = t_{\Box \Diamond \text{scheduled}(x)}(x)$ we get an implicit rankings with parameter x , which needs to be aggregated to get a closed implicit ranking.

The next constructor takes as input an implicit ranking Rk° and a formula α and returns an implicit ranking that refines the given implicit ranking such that states are ranked based Rk° only when α is satisfied, and states that do not satisfy α are ranked lower than those that do. It follows that every state that does not satisfy α is minimal according to this ranking. This constructor is useful when we can only guarantee conservation ($\leq$) of Rk° when α is satisfied.

Constructor 2. *The conditional constructor takes an implicit ranking $\text{Rk}^\circ = (\tau_{>}^\circ, \tau_{\geq}^\circ, \kappa_{\min}^\circ)$ with parameters $\mathbf{x}$ and soundness condition $\mathcal{C}^\circ$, and a formula $\alpha(\mathbf{x})$. It returns an implicit ranking $\text{Cond}(\text{Rk}^\circ, \alpha) = (\tau_{>}, \tau_{\geq}, \kappa_{\min})$ with parameters $\mathbf{x}$ and soundness condition $\mathcal{C}$ defined by:*

$$\begin{aligned}\tau_{>}(\mathbf{x}, \mathbf{x}') &= (\alpha(\mathbf{x}) \wedge \neg \alpha'(\mathbf{x}')) \vee (\alpha(\mathbf{x}) \wedge \alpha'(\mathbf{x}') \wedge \tau_{>}^\circ(\mathbf{x}, \mathbf{x}')) \\ \tau_{\geq}(\mathbf{x}, \mathbf{x}') &= \neg \alpha'(\mathbf{x}') \vee (\alpha(\mathbf{x}) \wedge \alpha'(\mathbf{x}') \wedge \tau_{\geq}^\circ(\mathbf{x}, \mathbf{x}')) \quad \kappa_{\min}(\mathbf{x}) = \neg \alpha(\mathbf{x}) \quad \mathcal{C} = \mathcal{C}^\circ\end{aligned}$$

Example 3. In Ex. 2 we show how to use the timer $t_{\Box \Diamond \text{scheduled}(x)}(x)$ in implicit rankings. In the ticket protocol every thread is scheduled infinitely often so the timer above is not always conserved. Moreover, a thread being scheduled does not necessarily change the state, for example, when it tries to enter the critical section but does not hold serv. For this purpose we want to consider the timer above only for threads that hold the currently served ticket. This is captured by the implicit ranking $\text{Cond}(\text{Pos}(t_{\Box \Diamond \text{scheduled}(x)}(x), <), \text{myt}(x) = \text{serv})$.

We continue by reintroducing the domain-based constructors of [21]. These aggregate rankings that are parameterized by domain elements. In [21] these were defined to be sound only for finite domains. Here, we slightly weaken this assumption and instead introduce it as part of the soundness condition, requiring finiteness only of the set of non-minimal elements in the base ranking. Since minimal values cannot decrease, having finitely-many non-minimal values is sufficient to maintain well-foundedness of the aggregated ranking.

The domain-pointwise constructor lifts an implicit ranking Rk° with ranking range $(P, <_P)$ to an implicit ranking whose ranking range is the set of functions

$Y \rightarrow P$ where Y is possibly infinite. The set $Y \rightarrow P$ is ordered by the pointwise ordering: $a <_{\text{pw}} b$ if and only if there exists $y \in Y$ such that $a(y) <_P b(y)$ and for every $y \in Y$ we have $a(y) \leq_P b(y)$. If $<_P$ is well-founded, we can restrict the produced partial order $<_{\text{pw}}$ to be well-founded even if Y is infinite by only considering functions $a \in Y \rightarrow P$ that have finitely many $y \in Y$ such that $a(y)$ is non-minimal. This is captured by the soundness condition of the constructor.

Constructor 3. The domain-pointwise constructor takes an implicit ranking $\text{Rk}^\circ = (\tau_{>}^\circ, \tau_{\geq}^\circ, \kappa_{\min}^\circ)$ with parameters $\mathbf{x} = \mathbf{y} \cdot \mathbf{z}$ and soundness condition $\mathcal{C}^\circ$. It returns an implicit ranking $\text{DomPW}(\text{Rk}^\circ, \mathbf{y}) = (\tau_{>}, \tau_{\geq}, \kappa_{\min})$ with parameters $\mathbf{z}$ and soundness condition $\mathcal{C}$ defined by:

$$\begin{aligned} \tau_{>}(\mathbf{z}, \mathbf{z}') &= \tau_{\geq}(\mathbf{z}, \mathbf{z}') \wedge (\exists \mathbf{y}. \tau_{>}^\circ(\mathbf{y} \cdot \mathbf{z}, \mathbf{y} \cdot \mathbf{z}')) \\ \tau_{\geq}(\mathbf{z}, \mathbf{z}') &= \forall \mathbf{y}. \tau_{\geq}^\circ(\mathbf{y} \cdot \mathbf{z}, \mathbf{y} \cdot \mathbf{z}') & \kappa_{\min}(\mathbf{z}) &= \forall \mathbf{y}. \kappa_{\min}^\circ(\mathbf{y} \cdot \mathbf{z}) \\ \mathcal{C} &= \{s \in \text{struct}(\Sigma) \mid s \in \mathcal{C}^\circ \text{ and } \forall v \in \text{assign}(\mathbf{z}). \text{finite}_{\mathbf{y}}(\neg \kappa_{\min}^\circ(\mathbf{y} \cdot \mathbf{z}), s, v)\} \end{aligned}$$

where $\text{finite}_{\mathbf{y}}(\alpha, s, v) := |\{u \in \text{assign}(\mathbf{y}) \mid (s, u \cdot v) \models \alpha(\mathbf{y} \cdot \mathbf{z})\}| < \infty$.

Example 4. The implicit rankings constructed in Ex. 3 captures the decrease of rank of a scheduling timer for some thread x if it holds the currently served ticket, it thus has x as a parameter. If we aggregate over all such threads with $\text{DomPW}(\text{Cond}(\text{Pos}(t_{\square \diamond \text{scheduled}}(x), <), \text{myt}(x) = \text{serv}), x)$ we consider the scheduling of all such threads. This pattern is the primary way we utilize timers for implicit rankings, and so we introduce “sugar” for it:

$$\text{TimerRank}(\psi(\mathbf{x}), \alpha(\mathbf{x})) = \text{DomPW}(\text{Cond}(\text{Pos}(t_{\psi}(\mathbf{x}), <), \alpha(\mathbf{x}), \mathbf{x})).$$

The final constructor lifts an implicit ranking Rk° with ranking range $(P, <_P)$ to an implicit ranking whose ranking range is the set of functions $Y \rightarrow P$ where $(Y, <_Y)$ is a possibly infinite well-founded set. The set of functions is ordered by the *reverse* lexicographic ordering: $a <_{\text{lex}} b$ if and only if for every $y \in Y$ such that $a(y) \not<_P b(y)$ there exists y_0 such that $y <_Y y_0$ and $a(y_0) <_P b(y_0)$. Well-foundedness of $<_Y$ along with finiteness of the set of non-minimal elements in Rk° ensure well-foundedness of the reverse lexicographic ordering. A reverse lexicographic ordering is crucial. To see this, consider for example the standard lexicographic ordering on $\{0, 1\}^{\mathbb{N}}$, which is not well-founded since it includes the sequence $(1, 0, \dots), (0, 1, 0, \dots), (0, 0, 1, 0, \dots), \dots$ that is infinitely decreasing.

Constructor 4. The domain-lexicographic constructor takes an implicit ranking $\text{Rk}^\circ = (\tau_{>}^\circ, \tau_{\geq}^\circ, \kappa_{\min}^\circ)$ with parameters $\mathbf{x} = \mathbf{y} \cdot \mathbf{z}$ and soundness condition $\mathcal{C}^\circ$, and a relation $\ell(y_1, y_2)$ over Σ . It returns an implicit ranking $\text{DomLex}(\text{Rk}^\circ, \mathbf{y}, \ell) = (\tau_{>}, \tau_{\geq}, \kappa_{\min})$ with parameters $\mathbf{z}$ and soundness condition $\mathcal{C}$ defined by:

$$\begin{aligned} \tau_{>}(\mathbf{z}, \mathbf{z}') &= \tau_{\geq}(\mathbf{z}, \mathbf{z}') \wedge (\exists \mathbf{y}. \tau_{>}^\circ(\mathbf{y} \cdot \mathbf{z}, \mathbf{y} \cdot \mathbf{z}')) \\ \tau_{\geq}(\mathbf{z}, \mathbf{z}') &= \text{order}(\ell) \wedge \forall \mathbf{y}. (\tau_{\geq}^\circ(\mathbf{y} \cdot \mathbf{z}, \mathbf{y} \cdot \mathbf{z}') \vee \exists y_0. (\ell(y, y_0) \wedge \tau_{>}^\circ(y_0 \cdot \mathbf{z}, y_0 \cdot \mathbf{z}')))) \\ \kappa_{\min}(\mathbf{z}) &= \forall \mathbf{y}. \kappa_{\min}^\circ(\mathbf{y} \cdot \mathbf{z}) \\ \mathcal{C} &= \left\{ s \in \text{struct}(\Sigma) \mid \begin{array}{l} s \in \mathcal{C}^\circ, \ell \text{ is well-founded in } s, \\ \forall v \in \text{assign}(\mathbf{z}). \text{finite}_{\mathbf{y}}(\neg \kappa_{\min}^\circ(\mathbf{y} \cdot \mathbf{z}), s, v) \end{array} \right\} \end{aligned}$$

Example 5. Consider a program that takes as input an array of natural numbers $c[0, \dots, n-1]$, and in each iteration, an index i with $c[i] > 0$ is chosen and the operations $c[i] := c[i] - 1$; $c[i-1] = *$ are performed. This program can be shown to be terminating with the implicit ranking $\text{DomLex}(\text{Pos}(c[i], <_{\mathbb{N}}), i, <_{\mathbb{N}})$.

5.3 Validating Decrease of Rank and Soundness Conditions

We now revisit Theorem 2 and show how to verify its two premises in order to prove termination. Fix a signature Σ , a transition system $\mathcal{T}(\mathcal{S})$ with $\mathcal{T} = (I, \iota, \tau)$ and a closed implicit ranking $\tau_{>}$ with soundness condition $\mathcal{C}$. We need to show that any reachable state of $\mathcal{T}(\mathcal{S})$ satisfies the soundness condition $\mathcal{C}$ and any reachable transition satisfies $\tau_{>}$. To discharge the premises we utilize an inductive invariant θ as a way to overapproximate the reachable states, for which we need to verify that θ is satisfied in the initial states and that θ is preserved by transitions. Then, for the second premise, we need to verify that transitions beginning in a state satisfying θ satisfy $\tau_{>}$. These requirements are captured by the validity of the following verification conditions in first-order logic: (i) $\iota \wedge \Gamma \rightarrow \theta$ (ii) $\theta \wedge \Gamma \wedge \tau \wedge \Gamma' \rightarrow \theta'$ (iii) $\theta \wedge \Gamma \wedge \tau \wedge \Gamma' \rightarrow \tau_{>}$.

As for the soundness condition $\mathcal{C}$, verifying that a certain structure s satisfies $\mathcal{C}$ is not always possible in first-order logic. This is because we use the soundness condition to encode those requirements of the ranking that are not definable in first-order logic. Specifically, the soundness conditions introduced by our constructors are of two types: well-foundedness of a relation and finiteness of a set. These notions are not first-order definable, so we cannot reduce their verification to first-order validity checks. Instead, we employ two complementary strategies to verify the soundness condition: relying on the intended system semantics, or finding sufficient conditions that are expressible in first-order logic.

Strategy 1. In some cases the intended system semantics trivially implies the validity of certain soundness conditions: 1. A set of elements of a sort with finite-domain semantics is finite. 2. A partial order on a sort with finite-domain semantics is well-founded. 3. A partial order that is well-founded in the intended semantics of the system, for example, the $<$ relation on a sort with intended semantics of the extended natural numbers (as in timers), is well-founded.

Strategy 2. The second strategy is aimed at showing finiteness of a set of elements S in all reachable states, even though the domain of the relevant sort is not finite. The approach is based on [22], with some generalization. The idea is that if there is $m \in \mathbb{N}$ such that initially there are at most m elements in B and any transition adds at most m elements to B then by induction, B is finite in all reachable states. This also implies that any set A such that $A \subseteq B$ in all reachable states is finite. In the soundness conditions we consider, the set A is parametrized by some $\mathbf{z}$ and for a given $\mathbf{z}$, it is defined as the set of assignments to $\mathbf{y}$ that satisfy a formula $\alpha(\mathbf{y} \cdot \mathbf{z})$, where our goal is to verify finiteness of A for any $\mathbf{z}$. We therefore encode the aforementioned idea by considering a formula $\beta(\mathbf{y} \cdot \mathbf{z})$ used as an inductively-finite over-approximation of $\alpha(\mathbf{y} \cdot \mathbf{z})$. This results in the following verification conditions, outlined for $m = 1$, for any $\mathbf{z}$:

- (i) $\alpha(\mathbf{y} \cdot \mathbf{z}) \wedge \theta \wedge \Gamma \rightarrow \beta(\mathbf{y} \cdot \mathbf{z})$; (ii) $\iota \wedge \theta \wedge \Gamma \rightarrow \forall \mathbf{z} \exists \mathbf{y}_0 \forall \mathbf{y} \beta(\mathbf{y} \cdot \mathbf{z}) \rightarrow \mathbf{y} = \mathbf{y}_0$; and (iii) $\theta \wedge \Gamma \wedge \tau \wedge \Gamma' \rightarrow \forall \mathbf{z} \exists \mathbf{y}_0 \forall \mathbf{y} \beta'(\mathbf{y} \cdot \mathbf{z}) \rightarrow \mathbf{y} = \mathbf{y}_0 \vee \beta(\mathbf{y} \cdot \mathbf{z})$.

Example 6. Continuing with Ex. 4, the soundness condition can be understood as the requirement that in all reachable states there are finitely many threads x for which $\alpha(x) = (\text{myt}(x) = \text{serv})$ holds. If we take $\beta(x) = \neg(\text{idle}(x))$ and an appropriate invariant θ we can easily verify the verification conditions of strategy 2 above with $m = 1$, and thus validate the soundness condition.

6 Implementation and Evaluation

We implemented our approach for verification of temporal properties with timers and implicit rankings in a deductive verification tool. Our tool is implemented in Python, using the Z3 API [23], it is available at [9]. It takes as input a specification of a transition system in first-order logic, along with a temporal property in FO-LTL, an implicit ranking given by a composition of constructors, an inductive invariant, and, when strategy 2 of Section 5.3 is applied, also inductively-finite approximations, all of which can use the timer variables. The tool computes the timer reduction of Section 4.2, generates the verification conditions of Section 5.3 and discharges them using the Z3 solver. When one of the verification conditions is violated, the tool produces a counterexample in the form of a transition of the augmented system. In the SMT queries, the timers sort σ_{time} is encoded using interpreted integers (with ∞ encoded by -1). The combination of uninterpreted functions, quantifiers and integers is sometimes difficult for the solver. We therefore implement an optimization where for invariants that do not involve timers we allow to check inductiveness w.r.t. the original system without the timers. Additionally, similarly to [21], we allow hint terms for existential quantifiers which help in queries with quantifier alternations.

We evaluated our tool on a set of examples of transitions systems and their properties from previous works [3, 15, 21, 22, 25], with user-provided implicit rankings and invariants. We expand on all examples in [20]. We ran our tool using a Macbook Pro with an Apple M1 Pro CPU, Z3 version 4.13.3.

Results. Table 2 summarizes our experimental results. To give a sense of the complexity of the proofs, we report for each example the time required for validating the verification conditions, the number of constructors used in the implicit ranking (where TimerRank is counted as one), the number of inductively-finite approximations needed, and the number of conjuncts in the invariant. We additionally report the total size of the proof, measured by the number of tokens (or terms). Where a proof by the liveness-to-safety method exists, we also report this same metric for the corresponding proof, taken from Ivy [27].

For examples from [21, 22], our proofs are similar to the original ones, except that in those works specific liveness proof rules were used while we prove termination of the augmented system. The modifications of the proofs to our general framework are relatively simple, converting premises of the aforementioned rule into instantiations of the TimerRank constructor. For examples that were proven

Table 2. Evaluation Results. Time is given in seconds, Con. stands for the number of constructors in the implicit ranking, Fin stands for the number of inductively-finite approximations, Inv. stands for the number of conjuncts in the invariant, |Proof| stands for the number of tokens in the entire proof and |L2S Proof| stands for the number of tokens in the entire proof with the liveness-to-safety reduction in the Ivy file.

Example	Source	Time (s)	Con.	Fin	Inv.	Proof	L2S Proof
Ticket (Section 2)	[25]	2	6	2	20	256	407
Paxos		101	11	1	20	238	521
ABP		4	31	2	34	539	860
HRB-Correct	[3]	135	7	0	6	99	446
HRB-Relay		56	8	0	17	316	593
Ackermann	[15]	0.8	5	2	9	211	748
LexArray (Ex. 5)	-	0.08	4	1	2	30	-
TimestampedQueue	[22]	0.5	6	1	2	73	-
CascadingQueue		1.2	9	2	7	175	-
ReorderingQueue		2.3	9	2	11	220	-
MutexRing	[21]	0.5	8	0	5	93	-
LeaderRing		0.3	8	1	4	76	-
ToyStabilization		0.3	5	0	3	69	-
Dijkstra k -State		64.8	6	0	3	213	-
BinaryCounter		0.02	4	0	0	21	-
SAT-Backtrack		0.3	9	0	8	157	-
SAT-CDCL		5.3	8	0	8	148	-

by the liveness-to-safety, our proofs are simpler, as they do not require reasoning about a finite abstraction or a monitored copy of the state. This can be observed in the significant difference in proof size in some examples.

7 Related Work

We have presented an approach for verifying temporal properties of systems that operate over unbounded domains, modeled in first-order logic. Many works are aimed at verifying similar systems and properties. Some works take a fully automatic approach that applies to restricted systems [2, 5–7, 10, 11, 14, 16, 28], notably these works do not apply to the setting of systems specified in first-order logic. Other works provide frameworks to design distributed systems and mechanically prove their liveness [4, 13, 17, 19, 24, 29–31, 35] and finally some works provide theoretical frameworks to analyze proof mechanisms for FO-LTL [1, 12, 34]. We turn to discuss the most relevant works, which, similarly to our work, provide semi-automatic methods that rely on first-order logic solvers.

Our work is based on [21] and extends it in two ways. First, while the original work targets only response properties with parameterized fairness assumptions, our work targets any FO-LTL property. Second, the domain-based aggregations of implicit rankings in the original work assume that the relevant domain is finite. In our work we allow domain-based aggregations on infinite domains, as long as the set of non-minimal elements of the aggregated rankings remains finite.

In [22] relational rankings are used to verify temporal properties. Relational rankings are a subset of implicit rankings that are lexicographic compositions of domain-pointwise rankings. Similarly to our work, this work also allows systems with infinite domains, and our strategy 2 in Section 5.3 is inspired by their enforcement of finiteness. Unlike our unified approach, [22] considers different temporal properties by using different proof rules for different types of properties, which is more complex and less general.

The approach of [33] is to semi-automatically synthesize polynomial ranking functions from integer variables in the state of the system. These variables can be from the specification itself, cardinalities of certain sets and fairness variables introduced by the user. These fairness variables are similar to the timers we introduce in Definition 1, and are the inspiration for timers, but they capture only specific types of fairness assumptions and are introduced manually, so they do not have formal guarantees. Like us, they allow the user to denote certain sorts as finitely interpreted, but their treatment of finiteness is less complete. While some examples are shared between the papers, they make some manual manipulations to fairness assumptions in non-trivial ways that significantly simplify the proofs.

In [15, 25, 26] a liveness-to-safety reduction is employed to reduce the verification of temporal properties to safety verification. The reduction performs a dynamic finite abstraction of the system. One then proves acyclicity of the abstracted system by providing an invariant for a monitored system where the current state is compared to a saved state modulo the abstraction. The liveness-to-safety reduction is sound but not complete, and requires employing temporal prophecy to extend its power. Moreover, writing invariants for the augmented system may be more complicated and less intuitive than giving implicit rankings.

Acknowledgement We thank the anonymous reviewers and Eden Frenkel for helpful comments. The research leading to these results has received funding from the European Research Council under the European Union’s Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). This research was partially supported by the Israeli Science Foundation (ISF) grant No. 2117/23, by a research grant from the Center for New Scientists at the Weizmann Institute of Science and by a grant from the Azrieli Foundation.

Data-Availability Statement The paper is accompanied by an artifact containing the software and data required to reproduce the results. A snapshot of the artifact corresponding to the version used in this paper is available at [9]. For reuse and future development, the source code is available at [8].

References

1. Abadi, M.: The power of temporal proofs. *Theor. Comput. Sci.* **65**(1), 35–83 (1989), [https://doi.org/10.1016/0304-3975\(89\)90138-2](https://doi.org/10.1016/0304-3975(89)90138-2)
2. Ben-Amram, A.M., Lee, C.S.: Ranking functions for size-change termination II. *Log. Methods Comput. Sci.* **5**(2) (2009), <http://arxiv.org/abs/0903.4382>
3. Berkovits, I., Lazic, M., Losa, G., Padon, O., Shoham, S.: Verification of threshold-based distributed algorithms by decomposition to decidable logics. In: *Computer Aided Verification - 31st International Conference, CAV 2019* (2019), https://doi.org/10.1007/978-3-030-25543-5_15
4. Brunel, J., Chemouil, D., Tawa, J.: Analyzing the fundamental liveness property of the chord protocol. In: *Formal Methods in Computer Aided Design, FMCAD 2018* (2018), <https://doi.org/10.23919/FMCAD.2018.8603001>
5. Chen, Y., Heizmann, M., Lengál, O., Li, Y., Tsai, M., Turrini, A., Zhang, L.: Advanced automata-based algorithms for program termination checking. In: *Programming Language Design and Implementation, PLDI 2018* (2018), <https://doi.org/10.1145/3192366.3192405>
6. Cimatti, A., Griggio, A., Magnago, E.: LTL falsification in infinite-state systems. *Information and Computation* (2022), <https://www.sciencedirect.com/science/article/pii/S0890540122001328>
7. Daniel, J., Cimatti, A., Griggio, A., Tonetta, S., Mover, S.: Infinite-state liveness-to-safety via implicit abstraction and well-founded relations. In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17–23, 2016* (2016), https://doi.org/10.1007/978-3-319-41528-4_15
8. Elad, N., Lotan, R.: implicit-rankings-timers, <https://github.com/neta-elad/implicit-rankings-timers>
9. Elad, N., Lotan, R.: Verifying first-order temporal properties of infinite states systems via timers and rankings (artifact) (Jan 2026), <https://doi.org/10.5281/zenodo.18157346>
10. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.D.: Liveness with invisible ranking. *Int. J. Softw. Tools Technol. Transf.* **8**(3), 261–279 (2006). <https://doi.org/10.1007/S10009-005-0193-X>
11. Farzan, A., Kincaid, Z., Podelski, A.: Proving liveness of parameterized programs. In: *Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16* (2016). <https://doi.org/10.1145/2933575.2935310>
12. Geatti, L., Gianola, A., Gigante, N.: First-order automata. In: *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence* (2025), <https://doi.org/10.1609/aaai.v39i14.33638>
13. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: proving safety and liveness of practical distributed systems. *Commun. ACM* **60**(7), 83–92 (2017), <https://doi.org/10.1145/3068608>
14. Herrmann, R., Rümmer, P.: A new approach for showing termination of parameterized transition systems. In: *Implementation and Application of Automata* (2026), https://doi.org/10.1007/978-3-032-02602-6_14
15. Hoenicke, J., Padon, O., Shoham, S.: On the Power of Temporal Prophecy, pp. 40–53. *Springer Nature Switzerland, Cham* (2026), https://doi.org/10.1007/978-3-032-13711-1_3
16. Hong, C., Lin, A.W.: Regular abstractions for array systems. *Proc. ACM Program. Lang.* **8**(POPL), 638–666 (2024), <https://doi.org/10.1145/3632864>

17. Ioannidis, E., Zakowski, Y., Zdancewic, S., Angel, S.: Structural temporal logic for mechanized program verification (2025), <https://arxiv.org/abs/2410.14906>
18. Lamport, L.: A new solution of dijkstra's concurrent programming problem. *Commun. ACM* **17**(8), 453–455 (1974), <https://doi.org/10.1145/361082.361093>
19. Leung, D., Zeldovich, N., Kaashoek, F.: Shipwright: Proving liveness of distributed systems with byzantine participants (2025), <https://arxiv.org/abs/2507.14080>
20. Lotan, R., Elad, N., Padon, O., Shoham, S.: Verifying first-order temporal properties of infinite-state systems via timers and rankings (2026), <https://arxiv.org/abs/2601.13325>
21. Lotan, R., Shoham, S.: Implicit rankings for verifying liveness properties in first-order logic. In: *Tools and Algorithms for the Construction and Analysis of Systems TACAS 2025* (2025). https://doi.org/10.1007/978-3-031-90643-5_20
22. McMillan, K.L.: Toward liveness proofs at scale. In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I. Lecture Notes in Computer Science*, vol. 14681, pp. 255–276. Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_13
23. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, (2008)*. https://doi.org/10.1007/978-3-540-78800-3_24
24. Ouyang, L., Sun, X., Tang, R., Huang, Y., Jivrajani, M., Ma, X., Xu, T.: Multi-grained specifications for distributed system model checking and verification. In: *Proceedings of the Twentieth European Conference on Computer Systems. EuroSys '25* (2025). <https://doi.org/10.1145/3689031.3696069>
25. Padon, O., Hoenicke, J., Losa, G., Podelski, A., Sagiv, M., Shoham, S.: Reducing liveness to safety in first-order logic. *Proc. ACM Program. Lang.* **2**(POPL), 26:1–26:33 (2018). <https://doi.org/10.1145/3158114>
26. Padon, O., Hoenicke, J., McMillan, K.L., Podelski, A., Sagiv, M., Shoham, S.: Temporal prophecy for proving temporal properties of infinite-state systems. *Formal Methods Syst. Des.* **57**(2), 246–269 (2021). <https://doi.org/10.1007/S10703-021-00377-1>
27. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: Safety verification by interactive generalization. In: *PLDI '16: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 614–630 (Jun 2016). <https://doi.org/10.1145/2908080.2908118>
28. Podelski, A., Rybalchenko, A.: Transition invariants. In: *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*. pp. 32–41. IEEE Computer Society (2004). <https://doi.org/10.1109/LICS.2004.1319598>
29. Qiu, L., Kim, Y., Shin, J., Kim, J., Honoré, W., Shao, Z.: Lido: Linearizable byzantine distributed objects with refinement-based liveness proofs. *Proc. ACM Program. Lang.* **8**(PLDI), 1140–1164 (2024), <https://doi.org/10.1145/3656423>
30. Qiu, L., Xiao, J., Shin, J.Y., Shao, Z.: Lido-dag: A framework for verifying safety and liveness of dag-based consensus protocols. *Proc. ACM Program. Lang.* **9**(PLDI) (Jun 2025), <https://doi.org/10.1145/3729306>
31. Sun, X., Ma, W., Gu, J.T., Ma, Z., Chajed, T., Howell, J., Lattuada, A., Padon, O., Suresh, L., Szekeres, A., Xu, T.: Anvil: Verifying liveness of cluster management controllers. In: *18th USENIX Symposium on Operating Systems Design and Implementation* (2024), <https://www.usenix.org/conference/osdi24/presentation/sun-xudong>

32. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Logics for Concurrency - Structure versus Automata (1995). https://doi.org/10.1007/3-540-60915-6_6
33. Yao, J., Tao, R., Gu, R., Nieh, J.: Mostly automated verification of liveness properties for distributed protocols with ranking functions. *Proc. ACM Program. Lang.* **8**(POPL), 1028–1059 (2024). <https://doi.org/10.1145/3632877>
34. Zhang, W.: First order büchi automata and their application to verification of LTL specifications. *J. Log. Algebraic Methods Program.* **142**, 101021 (2025). <https://doi.org/10.1016/J.JLAMP.2024.101021>
35. Zhao, Q., Pîrlea, G., Grzeszkiewicz, K., Gilbert, S., Sergey, I.: Compositional verification of composite byzantine protocols. In: Conference on Computer and Communications Security. CCS '24, Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3658644.3690355>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Revisiting Stateful Partial-Order Reduction

Frédéric Herbreteau , Gérald Point ,

Gautham Viswanathan , and Igor Walukiewicz

Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400 Talence, France
fh@labri.fr, point@labri.fr, viswanathangautham@gmail.com, igw@labri.fr

Abstract. Partial-order methods accelerate the verification of concurrent programs. These methods are based on a concept of *independence* of actions, which induces an *equivalence* between different program runs. The equivalence allows for the construction of a *reduced transition system* that represents all runs up to this equivalence. Such a system can be orders of magnitude smaller than the full transition system representing all possible runs, while still containing all relevant information.

We propose a new partial-order reduction method for concurrent acyclic programs that use variables and locks. Our algorithm is derived from first principles, suggesting it can be readily adapted to other settings. As our evaluation demonstrates, the algorithm offers substantial gains over existing partial-order methods.

1 Introduction

The goal of partial-order methods is to speed up explicit state exploration of concurrent systems. The state space of such systems grows exponentially with the number of processes. Fortunately, many runs of a concurrent system are often equivalent, making it sufficient to explore only one run from each equivalence class. For example, if one process sets x to 2 and another sets y to 3, the order of execution is usually irrelevant. The two execution sequences are equivalent, and it is sufficient to explore only one of them. This reduces both the number of visited states and the time required for exploration. In some cases, these reductions can be substantial.

Partial-order methods offer more than just efficient state-space exploration. An important application is proving properties of concurrent programs [15], where equivalence between runs is used not only to reduce the number of proof objectives but also to simplify them by choosing particular linearizations. In symbolic execution [13] and testing [35], partial-order methods can limit the exploration space while remaining exhaustive. Another application lies in the verification of timed systems using recently developed local-time zones [22]. Finally, a newly published report [8] describes how AWS uses an explicit-state model checker with a simple partial-order reduction to analyze the achievability

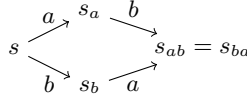


Fig. 1. Diamond property: if both a and b are enabled in state s , then executing them in either order leads to the same state.

of latency service-level agreements (SLAs) and other system behaviors beyond standard reachability or liveness properties. All these applications rely on explicit state enumeration, as opposed to symbolic methods such as SAT or BDDs.

The first partial-order methods - known as stubborn sets, persistent sets, and ample sets - were proposed about 35 years ago [44,21,28]. Distinctions between the three [46] are not important for this discussion, so we will focus on persistent sets. These initial works sparked a continuous stream of research on the topic. In 2005, Flanagan and Godefroid introduced *stateless* dynamic reductions [18], making stateless methods a focal point of subsequent research. Their starting point was an observation that many insights can be gained already from bounded model-checking, or in other words, analysis of acyclic programs. Indeed, in the last decade a great majority of works in the field restricts to acyclic programs also because the methods of treating cycles are mostly orthogonal to a partial-order method employed, and essentially the same since the initial works on the subject [44,21,28]. In 2014, Abdulla et al. proposed a notion of trace-optimality [1] and a race reversal technique, initiating a new stage of work on stateless partial-order methods [10,5,26,30,31,3,29].

In contrast to this trend, we focus on *stateful* partial-order reduction methods, which have seen relatively less progress over the last two decades [36,49,12,11]. The goal of a stateful partial-order algorithm is to produce a *reduced transition system* containing a representative for every equivalence class of runs. The *stateless* approach constructs a tree of runs and saves memory by storing only the current branch. This is essential when the states are too large to store them in memory. In contrast, the *stateful* approach stores all visited states, stopping exploration when a state is revisited. This produces a complete graph of the reduced system, which can then be analyzed by other tools.

Each approach has its advantages. When the global transition system has a tree-like structure and the objective is to verify reachability, stateless exploration is preferable. It may even be the only viable option when the state is too large or difficult to capture, as is the case when analyzing C multithreaded programs by executing the code directly. However, in many cases, such as when verifying pseudocode models, the reduced transition system may be of a reasonable size but contain an astronomical number of non-equivalent runs, rendering the stateless approach infeasible. For applications beyond simple reachability testing, such as those mentioned earlier, the stateful approach is also often preferable.

The starting point of our development is the notion of a diamond between two actions, as shown in Figure 1. It captures the principle that the execution

order of two actions is irrelevant. This notion is state-dependent. For example, if the current value of x is 1 then the actions $write_p(x, 2)$, namely process p writing 2 to x , and $await_q(x, [1, 3])$, namely process q waiting for x to be in the interval $[1, 3]$, can be executed in any order. However, in a state where x is 0, the order is critical, since $await_q(x, [1, 3])$ would be blocked. We can make it state independent by requiring that the actions should commute in every state where they are both enabled. This gives a notion of a *strong diamond*. For example, the actions $write_p(x, 2)$ and $await_q(x, [1, 3])$ are in the strong diamond relation. It turns out that a partial-order reduction method can be derived based solely on this notion of a strong diamond. Unfortunately, as it is the case with the persistent set method, we have proved that the resulting algorithm is inefficient because it relies on an NP-complete oracle test [23]. In this paper, we thus propose a heuristic that yields an efficient partial-order reduction method. To do this, we first observe that for any stateful method to be tractable, the set of variable values must be reasonably small. This assumption makes it feasible to analyze the state space of each process separately and use the results to guide the partial-order reduction in a much more refined way than the persistent set method. In our experiments, this pre-computation typically takes a fraction of a second, rising to a maximum of two seconds only for a particularly challenging example. The result is a partial-order method that is essentially a DFS exploration of the state space, with an additional computation performed at each state to determine which actions to explore next. Crucially, the cost of this computation depends on the size of the concurrent program, and not on the size of the global state space being constructed. It must be highly efficient, as it is executed at every state, and the number of states can be orders of magnitude larger than the program's size. We obtain a method that is not only theoretically superior to the state-of-the-art persistent set approach but also performs significantly better on our test examples.

In summary, our principal contributions are:

- A contextual notion of independence derived from the principle of a strong diamond; Section 4. This yields a contextual equivalence between runs that while building on existing ideas aims at balancing generality and usability. For example, it has bubbling property (Lemma 2) that does not hold for a contextual equivalence in general (e.g. Definition 3.17 in [21]).
- An efficient heuristic to implement the generic method, which approximates its NP-complete oracle test by pre-analyzing the state space of individual processes; Section 5.
- An implementation of our heuristic and an evaluation on standard models from the literature. It shows that on this type of models we obtain substantial performance gains compared to other stateful/stateless methods, and symbolic methods such as IC3, BDDs, and Dartagnan [19]; Section 6.

2 Related work

The literature on partial-order methods has expanded rapidly in the last decade. This discussion focuses on closely related results, particularly on recent work.

Partial-order methods were introduced in the early 1990s [44,20,28]. The fundamental principles from these seminal works remain the basis for modern stateful applications [16]. Among these, the concept of sleep sets [21] is particularly relevant to our work. We also draw upon the use of lexicographic ordering to identify representative runs [48,27,47]. In contrast, approaches based on Petri-net unfolding [40,12] are distinct from ours, as they rely on prime event structures rather than transition systems.

The independence relation derived from a strong diamond is a special case of conditional independence from [28], and from Chapter 3.4 of Godefroid's PhD thesis [21]. The latter is too general for us, because it does not satisfy bubbling lemma, see Lemma 2. The rest of our development follows a more modern approach of thinking about partial-order reduction; namely equivalence between runs and source sets [1].

The majority of recent research has focused on stateless partial-order reduction. First introduced by [18], this line of work gained significant momentum with the proposal of source-sets [1]. This research has culminated in *truly stateless* algorithms that require only polynomial-sized memory [30,3]. Very recently, these techniques have been extended to handle some blocking synchronization mechanisms [26,31]. In our work we handle standard blocking instructions like *await* and lock operations.

The race reversal technique introduced in [18,1] has been partially adapted for use in stateful methods [49,43,11]. There are important technical obstacles in implementing race reversal in a stateful setting, so these works simplify the method substantially, or employ several passes through the constructed graph with a potential exponential time complexity. Notably [43] considers programs with loops. The unfolding based partial-order methods have seen developments like the *k*-partial alternatives [12]. On the application front, stateful partial-order methods have been notably used to prove the correctness of concurrent programs [16,15], as well as for symbolic executions and testing [13,35].

A significant direction of research involves developing coarser equivalence relations than the classical Mazurkiewicz traces [34,10,5,16,29,39,17,6]. Our equivalence is also strong and contextual, and we believe the underlying principles can be extended to other frameworks. However, our specific heuristic is tied to the program syntax we define. Another fruitful research area has been the adaptation of partial-order methods to weak memory models [4,32,19,30]. In this work, we assume the standard sequential consistency model.

3 Concurrent programs

We consider a simple standard model of concurrent programs with variables and locks. Variables range over some fixed finite domain, and locks are binary. The only operation on variables are *write* and *await*.

A *process* is a directed acyclic graph with edges labeled by actions. These actions come from the set:

$$\Sigma ::= w_p(x, i) \mid a_p(x, I) \mid acq_p(\ell) \mid rel_p(\ell)$$

where p is a process name, x a variable, i a value, I a set of values, and ℓ a lock. Action $w_p(x, i)$ sets the value of x to i ; $a_p(x, I)$ blocks until the value of x is in I ; $acq_p(\ell)$ acquires the lock ℓ ; and $rel_p(\ell)$ releases the lock ℓ . A process must be *deterministic*, i.e., all outgoing actions from a state must have different labels, but these labels need not be mutually exclusive.

The restriction to acyclic programs is a common assumption in most recent works on partial-order reduction techniques (e.g. [1,10,5,26,30,31,3,29]). We discuss extension to programs with loops in the conclusion. Since we do not allow cycles in processes, it is quite natural to have a blocking *await* instruction. For example, all classical mutual exclusion algorithms rely on it. If we allowed cycles, *await* could be simulated by a while loop with a read. A conditional instruction is modeled by branching. A read operation can be simulated by branching between *await* operations.

A *concurrent program* $\mathbb{P}$ is a finite set of processes, over a finite set of variables and locks. Each variable ranges over the same finite domain. The value of a lock is either 0 (free), or 1 (locked). The semantics of a program $\mathbb{P}$ is given by a *transition system* $\text{TS}(\mathbb{P})$. A state s of $\text{TS}(\mathbb{P})$ contains a state s_p for each process $p \in \text{Proc}(\mathbb{P})$ and a value $s(x)$ for each variable x , as well as $s(\ell)$ for each lock ℓ . In the initial state, every process is in its initial local state, and every variable and lock have value 0. The transitions in $\text{TS}(\mathbb{P})$ are defined according to the usual semantics of instructions from Σ . For example, $s \xrightarrow{a_p(s, I)} s'$ if the value of x is in I , i.e., $s(x) \in I$, and $s' = s$. If $s(x) \notin I$, then $a_p(s, I)$ is not *enabled* in s . Write actions are always enabled. Each action comes with a subscript to indicate the process that performs it, so we can have both $a_p(x, I)$ and $a_q(x, I)$ enabled in s . We write *enabled*(s) for the set of enabled actions in s .

Observe that $\text{TS}(\mathbb{P})$ is acyclic since every process is acyclic. We require that every $rel(\ell)$ operation is preceded by an $acq(\ell)$ operation. This way we exclude programs where one process acquires a lock and another process releases it. This assumption removes some cases to consider, and reflects a common programming style. It implies that a release action is always enabled, namely if in a state s a process can do $rel(\ell)$ action then $s(\ell) = 1$.

We will need to talk about runs in a transition system $\text{TS}(\mathbb{P})$ as well as about local runs in a graph of each process. A *local run* in a process p is a path in the graph of p . We write $s_p \xrightarrow{v}_p s'_p$ to denote such a run between local states s_p and s'_p . A *global run*, or simply a run, in $\text{TS}(\mathbb{P})$ is a path not necessarily starting in the initial state. We write $s \xrightarrow{u} s'$ if u is a label of a path from s to s' . Sometimes we just write $s \xrightarrow{u}$ if s' is not relevant. Observe that if $s \xrightarrow{u} s'$ then the projection $v = u|_p$, that is the subsequence of actions from process p in u , is a local run $s_p \xrightarrow{v}_p s'_p$ for every process $p \in \text{Proc}(\mathbb{P})$. A *maximal run* is a run reaching a *terminal state* – a state with no outgoing transitions. A *full run* is a maximal run starting in the initial state. Thanks to our determinism

assumption, a label u determines the path from s and the final state s' , if it exists. We will write $u(s)$ for the state reached from s on the run u : $s \xrightarrow{u} u(s)$.

4 Introducing Partial-Order Reduction

Partial-order reduction is a technique for reducing the number of states of a transition system. The reduction is based on the idea of *independence* of actions inducing the notion of *equivalence* of runs. The aim is to construct a *reduced transition system* representing all the runs up to equivalence.

The novelty in this section is a nonstandard definition of independence. While the definition is derived from basic principles, we concentrate on our concurrent programs with four types of actions we have introduced. Another nonstandard aspect of our approach is that independence is indexed by a state. Yet, it is easier to work with than the general contextual independence [21].

4.1 Strong Diamond, Independence, and Equivalence

The notion of diamond property as in Figure 1 is the starting point of our approach. We show how to derive the notion of independence from it, and prove some basic properties of the derived independence relation.

Definition 1 (Strong diamond pair). *Two actions are in strong diamond relation if they commute in every state where they are both enabled, meaning $ab(s) = ba(s)$ whenever $a(s)$ and $b(s)$ exist. We write $\text{StrongDiamond}(a, b)$ for the strong diamond pair relation between actions a and b .*

For our actions Σ , we give a concrete definition of strong diamond pairs.

Lemma 1. *Two actions a, b of two different processes p, q are a strong diamond pair when either they use different variables/locks, or*

- $a = a_p(x, I)$ and $b = a_q(x, J)$, and I, J are arbitrary;
- $a = a_p(x, I)$ and $b = w_q(x, j)$, and $j \in I$;
- $a = w_p(x, i)$ and $b = w_q(x, j)$, and $i = j$;
- $a = acq(\ell)$ and $b = rel(\ell)$.

Strong diamond relation is not the same as independence relation. For example, acquire and release operations are in the strong diamond relation, but of course we will not assume that they are independent. Checking if a pair is strong diamond is usually easy, if the two actions share a process it is enough to check if they are strong diamond in the graph of this process. Strong diamond pairs allow us to define independence of actions in a given state, and an equivalence relation on runs from a given state.

Definition 2 (Independence). *Two actions a, b are independent in a state s , denoted aI_sb , if both a and b are enabled in s and $\text{StrongDiamond}(a, b)$ holds. We write aD_sb if not aI_sb ; we refer to D_s as the dependence relation.*

Example 1. Observe that unlike in the standard definition of independence, two writes are independent if they write the same value. Also, a write can be independent of await. For example $w_q(x, 1)$ is independent of $a_r(x, [0, 3])$ in a state where $x = 0$. On the other hand, $acq(\ell)$ is not independent of $rel(\ell)$ because the two are never simultaneously enabled in the same state.

Definition 3 (Equivalence relation on runs). *The equivalence relation $\sim_s$ on runs from s is the smallest reflexive and transitive relation such that $u_1 \sim_s u_2$ if there are decompositions $u_1 = vabv'$ and $u_2 = vbav'$ with $aI_{v(s)}b$.*

The definition of a *first action* in a sequence of actions is an important concept in partial-order reduction. An action b is a first action of a run when b starts a sequence equivalent to the run. In our case this definition depends on the state.

$$first(s, w) = \{b : \exists v. bv \sim_s w\} .$$

Example 2. Continuing Example 1, $first(s, w_q(x, 1)a_r(x, [0, 3]))$ consists of both actions if $x = 0$ in s , but only of the first action if $x = 5$ in s .

Our definitions are set up so that we can get every first action of a sequence by just permuting it up to the front of the sequence. It should be clear that without such a property we would have big difficulties in implementing a partial-order reduction. It is interesting to note that the property holds for every equivalence relation on sequences derived from strong diamond pairs, but it does not hold for all contextual equivalences (e.g. Definition 3.17 in [21]).

Lemma 2 (Bubbling). *Suppose $s \xrightarrow{ueu'}$ and $e \in first(s, ueu')$. For every decomposition $u = u_a au'_a$, with u_a a prefix of u , we have $aI_{u_a(s)}e$.*

4.2 Partial-Order Reduction

The equivalence relation on runs allows us to define a notion of a reduced transition system. In this section we will also introduce two very important notions: sleep sets [21] and source sets [1]. These allow us to formulate a generic partial order algorithm based on an *includes first set* oracle [23], or IFS test for short.

Recall that a concurrent program $\mathbb{P}$ defines a transition system $TS(\mathbb{P})$. This determines an independence relation I_s and an equivalence relation $\sim_s$; for every state s . In this section we keep $TS(\mathbb{P})$ fixed, so that we do not need to write it explicitly in our notation.

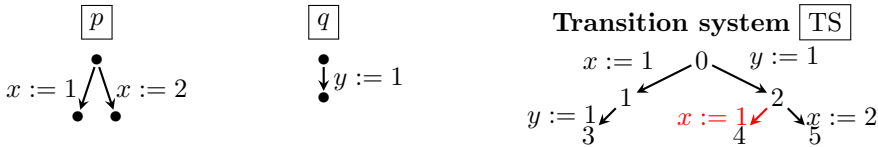
Definition 4 ([23]). *We say that rTS is a reduced transition system for $\mathbb{P}$ if:*

- *it is sound: every full run of rTS is a full run of $TS(\mathbb{P})$, and*
- *and complete: for every full run u of $TS(\mathbb{P})$ there is a full run u' of rTS such that $u \sim_{s^0} u'$; where s^0 is the initial state of $TS(\mathbb{P})$.*

Source sets [1] offer a way to think about how to construct a reduced transition system. For example if b is a first action of every run from s then $\{b\}$ is a source set, in a sense that it is enough to explore b from s . To make this intuition precise we start by defining the set of sets of first actions for all maximal runs from s : $First(s) = \{first(s, u) : u \text{ is a maximal run from } s\}$.

We say that F is a *covering source set* for s if F intersects every set in $First(s)$. For a given state s , let $out(s)$ denote the set of outgoing actions from s . We say that s is *covering* if $out(s)$ is a covering source set for s . This notion is important as it can guarantee completeness. Say we have a program $\mathbb{P}$ and a transition system TS such that every state s is covering. Then it is not hard to see that TS is complete.

Example 3. Not all states in a complete transition system need to be complete. In TS below, the set $\{x := 2\}$ is not a covering source set in node 2. However, the transition system is still complete without the red edge $2 \xrightarrow{x:=1} 4$.



Sleep sets aim at removing the inefficiency illustrated in the example above. The idea of a sleep set is to avoid exploring runs starting from actions that have already been explored in a larger context. In our example above, node 2 would have a sleep set $\{x := 1\}$, because actions $x := 1$ and $y := 1$ are independent in 0 and $x := 1$ was explored before $y := 1$. In general, in a transition system that is a tree with some creation order $\leq_{cr}$ on siblings, *sleep sets* are defined by the following rules:

- The sleep set of the initial state s^0 is empty: $sleep(s^0) = \emptyset$.
- If s is a state and $s \xrightarrow{e} s'$ then $sleep(s')$ is defined from $sleep(s)$ by adding all actions explored from s before e , and remove all actions dependent on e :

$$sleep(s') = (sleep(s) \cup \{b \in out(s) : b \leq_{cr} e\}) - D_s e$$

Instead of asking that $out(s)$ is a covering source set, we can ask that $out(s) \cup sleep(s)$ is. This is enough to ensure that the constructed transition system is complete. Unfortunately, this approach may produce *sleep blocked* runs, that end in a state where all enabled actions are in the sleep set. Such runs are a source of inefficiency as the algorithm produces a part of the transition system that turns out to contain only representatives of runs that have already been explored. Following [1] avoiding sleep blocked runs boils down to finding a *trace optimal* reduced transition system, that is one not containing two equivalent runs. In the context of stateless partial-order reduction, this notion is simply called *optimal*. In our case, the notion does not imply that the number of states in the reduced transition system is minimal, but it is a good guiding principle.

We propose to use what we call an “includes first set” oracle, abbreviated as *IFS*, as an abstract tool for avoiding sleep blocking.

Definition 5 ([23]). Let s be a state of $\text{TS}(\mathbb{P})$ and B a subset of actions. We say that B includes a first-set in s if there is a maximal run u from s with $\text{first}(s, u) \subseteq B$. Then, we say that $\text{IFS}(s, B)$ holds.

The idea behind IFS is as follows. Suppose an algorithm has reached a state s and produced a sleep set $\text{sleep}(s)$ containing actions that need not be explored from s . We would like to check if there is something left to be explored. We need to check if the exploration is not sleep-blocked, that is, if there is a maximal run u from s , with $\text{first}(s, u) \cap \text{sleep}(s) = \emptyset$. This can be done with predicate $\text{IFS}(s, \text{enabled}(s) - \text{sleep}(s))$. If the test holds, we need to explore from s , otherwise we do not have to consider s .

A partial-order reduction algorithm based on IFS is presented in Listing 1.1 (inspired from Listing 2 in [23]). It is a DFS exploration where every node consists of a state and a sleep set. In a node n the algorithm considers successors n' of n . If a node n' on this list is *subsumed* by some already visited node, denoted $n'' \triangleleft n'$, then it is not explored. The *subsumption* means that the states in the two nodes are the same and the sleep set in n' includes that of n'' . Then every run from n' also exists from n'' . The main point of the algorithm is to not explore successors that only yield sleep-blocked runs. This is ensured by the $\text{PIFS}(s, B)$ test (line 14), a heuristic approximating the $\text{IFS}(s, B)$ test. We need to use a heuristic as $\text{IFS}(s, B)$ test is NP-hard [23]. We explain PIFS in the next section.

5 PIFS

Our objective is to develop an efficient, approximation of includes a first-set test that does not hamper the correctness of the algorithm from Listing 1.1. This means that the approximation can have only a one-sided error. If the heuristic incorrectly returns True, the algorithm remains correct but simply performs a less effective reduction. However, if it incorrectly returned False, the algorithm would erroneously prune essential execution paths, compromising correctness. Therefore, the heuristic must guarantee that a False result is always accurate. To state this more formally, we define an $\text{IFS}(s, B)$ -run as a run u starting from s such that its initial actions are contained in B (i.e., $\text{first}(s, u) \subseteq B$). The central challenge is to design a heuristic that can certify the non-existence of a *maximal* $\text{IFS}(s, B)$ -run without resorting to an exhaustive search in the global state space. Definition 9 together with Theorem 1 present such a test.

In order to certify that there is no maximal $\text{IFS}(s, B)$ -run we need to somehow say that an $\text{IFS}(s, B)$ -run would not be able to involve some enabled process that is not in B . For this we will over-approximate what writes, and initial await instructions can happen on $\text{IFS}(s, B)$ -runs. This comes in a form of two sets $\text{AIR}(s, B)$ and $\text{APW}(s, B)$:

$$\begin{aligned} \text{AIR}(s, B) &= \bigcup \{ \text{IR}(u) : s \xrightarrow{u}, \text{first}(s, u) \subseteq B \} \\ \text{APW}(s, B) &= \bigcup \{ \text{PW}(u) : s \xrightarrow{u}, \text{first}(s, u) \subseteq B \} \end{aligned}$$

Listing 1.1. Explore using *PIFS* [23]

```

1  main( $\mathbb{P}$ ):
2    create node  $n^0$  with  $s(n^0) = s^0$  and  $sleep(n^0) = \emptyset$ 
3     $Explored := \emptyset$ 
4     $Explore(n^0)$ 
5
6  procedure  $Explore(n)$ :
7     $Sleep := sleep(n)$ 
8    while  $enabled(n) - Sleep \neq \emptyset$ :
9      let  $b \in (enabled(n) - Sleep)$ 
10     let  $s'$  such that  $s(n) \xrightarrow{b} s'$ 
11      $Sleep' := Sleep - D_s b$ 
12     if  $\exists n'' \in Explored$  s.t.  $n'' \triangleleft (s', Sleep')$ 
13       add edge  $n \xrightarrow{b} n''$  to the exploration graph
14     else if  $PIFS(s', enabled(s') - Sleep')$ 
15       create node  $n' = (s', Sleep')$ 
16       add edge  $n \xrightarrow{b} n'$ 
17        $Explore(n')$ 
18        $Sleep := Sleep \cup \{b\}$ 
19   add  $n$  to  $Explored$ 

```

where $IR(u)$ is the set of awaits $a_p(x, I)$ that appear before any write to x on u , and $PW(u)$ is the set of all writes $w_p(x, i)$, $acq_p(\ell)$ and $rel_p(\ell)$ on u . Observe that acquire and release actions are treated as writes. Indeed, both modify the value of a lock variable.

Given the two sets $APW(s, B)$ and $AIR(s, B)$ we can check if some action b of a process not in B , can be scheduled, meaning that b can appear in some extension of an $IFS(s, B)$ -run but not as a first action of this extension. However, it is difficult to compute $AIR(s, B)$ and $APW(s, B)$ precisely. So we will present a test guaranteeing that some pair (AIR, APW) is an over-approximation of $AIR(s, B)$ and $APW(s, B)$.

We first introduce some abbreviations for conciseness:

- $AIR(x) = \{I : a_p(x, I) \in AIR, \text{ for some } p\}$, all sets in awaits on x in AIR ;
- $APW(x)$ contains all i such that $w_p(x, i) \in APW$, for some p ;
- $APW(\ell)$ contains 1 if $acq_p(\ell) \in APW$, and 0 if $rel_p(\ell) \in APW$, for some p .

Definition 6. We say that a local transition $s_p \xrightarrow{b}$ of a process p is scheduled in s by (B, AIR, APW) if either $b \in B$, or b satisfies one of the following:

- $b = a_p(x, I)$, $s(x) \notin I$ and $\exists i \in APW(x)$. $i \in I$; or
- $b = a_p(x, I)$, $s(x) \in I$ and $\exists i, j \in APW(x)$. $i \in I$ and $j \notin I$; or

- $b = w_p(x, i)$ and $\exists j \in APW(x). j \neq i$, or $\exists J \in AIR(x). s(x) \in J, i \notin J$; or
- $b = acq_p(\ell)$ and $0 \in APW(\ell)$.

Suppose that we have discovered a run u with writes in APW and initial requirements in AIR . In this context $b \notin B$ is scheduled means that b can appear in some extension of u but not as a first action of this extension. Indeed, if $b = a_p(x, I)$ and b is not enabled in s , then b can appear in some extension of u if there is a write $w_q(x, i) \in APW$ enabling it (this is the first condition of the definition). If $b = a_p(x, I)$ and it is enabled in s , $s(x) \in I$, then in order for b not to be the first action there should be some write $w_q(x, j) \in APW$ blocking it, $j \notin I$, and some write $w_r(x, i) \in APW$ enabling it, $i \in I$. The conditions for other operations are derived by a similar reasoning.

It remains to compute over-approximating sets (AIR, APW) and to implement the test $PIFS(s, B)$ based on these two sets. This test will use information about local runs that we call signatures.

A signature of a local run says what await, write and lock actions happened on the run, forgetting the order between these actions. The only exception is that signature remembers initial requirements, namely await operations on a variable that are not preceded by a write to this variable.

Definition 7. For a local run v of a process p we define the signature $sig(v)$ as a triple $(IR(v), NR(v), PW(v))$:

- $IR(v)$ is the set of initial requirements, namely all actions $a_p(x, I)$ on v not preceded by a write to x .
- $NR(v)$ is the set of non-initial requirements, namely, all actions $a_p(x, I)$ on v preceded by a write to x .
- $PW(v)$ is the set of all produced writes, namely actions of the form $w_p(x, i)$, $acq_p(\ell)$, or $rel_p(\ell)$ on v .

We denote by $sig(s_p, b)$ the set of signatures of all local runs from state s_p of process p starting with b , i.e., $sig(s_p, b) = \{sig(bv) : s_p \xrightarrow{bv}_p\}$. We may assume that the states of processes are disjoint, so s_p identifies p uniquely.

Observe that we consider all local runs from s , not only the maximal ones.

Example 4. For a sequence

$$v = a_p(x, [1, 2]) \cdot w_p(x, 3) \cdot a_p(x, [4, 5]) \cdot w_p(x, 6) \cdot a_p(y, [7, 8])$$

we have: $IR(v) = \{a_p(x, [1, 2]), a_p(y, [7, 8])\}$ and $NR(v) = \{a_p(x, [4, 5])\}$ and $PW(v) = \{w_p(x, 3), w_p(x, 6)\}$. Observe that the signature does not remember the order between the actions, except that an action in $IR(v)$ comes before any write to the same variable.

The next definition is the main concept behind our heuristics for $PIFS(s, B)$ test. Assume that we have a pair (AIR, APW) saying that there may be a run u with $first(s, u) \subseteq B$, $IR(u) \subseteq AIR$ and $PW(u) \subseteq APW$. Now, we want to

check if u can be prolonged by some actions of some process p . We ask how a projection of this prolonged run on p may look like. This hypothetical local run of p must start with an action b scheduled by (B, AIR, APW) , thus either b is already part of u or it will be ordered after some action of u as discussed after Definition 6. This local run should require only awaits that can be provided by writes that are in u already, or that are initially satisfied by the valuation in s . Such a local run of p is approximated by its signature. So we look for $(IR, NR, PW) \in sig(s_p, b)$ such that for every non-initial await $a_p(x, I) \in NR$ we have a corresponding write $w_q(x, i) \in APW$ with $i \in I$, and for every initial await $a_p(x, I) \in IR$, either we have a corresponding write as above, or $s(x) \in I$.

Definition 8. We say that (AIR, APW) is a B -covering in s when for every process p , every transition $s_p \xrightarrow{b}$ scheduled in s by (B, AIR, APW) and every $(IR, NR, PW) \in sig(s_p, b)$ such that

- for every $a_p(x, I) \in NR$ there is $w_p(x, i) \in PW$ with $i \in I$, and
- for every $a_p(x, I) \in IR$ there is $w_p(x, i) \in PW$ with $i \in I$, or $s(x) \in I$

we have $IR \cap val(s) \subseteq AIR$ and $PW \subseteq APW$, where $IR \cap val(s)$ is the set of $a_p(x, I) \in IR$ such that $s(x) \in I$.

The essence of the *PIFS* test is to compute the smallest B -covering pair (AIR, APW) in s , and test if it “blocks” all actions enabled in s . Then, it is possible that there is a maximal *IFS*(s, B)-run, where blocking condition corresponds to the maximality requirement. The smallest B -covering pair exists because the closure property in Definition 8 is monotone componentwise, so the smallest covering can be computed iteratively starting from the pair of $(\emptyset, \emptyset)$.

Definition 9. The test *PIFS*(s, B) holds if for the smallest B -covering pair (AIR, APW) in s and every action b enabled in s we have that either:

- $proc(b) \in proc(AIR) \cup proc(APW)$, or
- $b = a_p(x, I)$ and $APW(x) \not\subseteq I$, or
- $b = acq(\ell)$ and $1 \in APW(\ell)$.

where $proc(AIR) = \{p : a_p(x, I) \in AIR, \text{ for some } x \text{ and } I\}$ is the set of processes that appear on an action in AIR , and similarly for $proc(APW)$.

Theorem 1. If *IFS*(s, B) holds then *PIFS*(s, B) holds.

This theorem says that the *PIFS* error is on the side that we wanted. Indeed, in the algorithm from Listing 1.1 we use *PIFS* in line 14 to check if it is worth to explore from a state s' . The theorem says that when using *PIFS* instead of *IFS* the condition may be true more often, so we may explore more states, but the resulting reduced transition system will be sound and complete.

We turn the definitions in this section into an algorithm for computing *PIFS* presented in Listing 1.2. For conciseness, we use the following notations. For a set W of writes, $W \models a_p(x, I)$ if there is a write $w_q(x, i) \in W$ such that $i \in I$,

for some q . We write $val(s) \models a_p(x, I)$ if $s(x) \in I$. Then, $W \cup val(s) \models a_p(x, I)$ if either $W \models a_p(x, I)$ or $val(s) \models a_p(x, I)$. Finally, given a set R of awaits, we write $W \cup val(s) \models R$ if $W \cup val(s) \models a_p(x, I)$ for all $a_p(x, I) \in R$.

The main task is to compute the smallest B -covering pair as in Definition 8. This can be done with a fixed point computation finding the least pair (AIR, APW) satisfying the closure condition. Starting from empty AIR and APW , the algorithm looks for an action b that is scheduled (line 7), as in Definition 6. If it finds such an action then for all signatures (IR, NR, PW) of $sig(s_p, b)$ it checks if something can be added to AIR or APW . This process continues until nothing can be added to AIR or APW in a sense of Definition 8. At this stage we know that (AIR, APW) is the smallest pair B -covering in s . The procedure returns the result of $FinalTest(s, AIR, APW)$ that implements the test from Definition 9.

Listing 1.2. PIFS test algorithm.

```

1  function PIFS( $s, B$ )
2     $AIR := APW := \emptyset$ ;
3     $P := \emptyset$  // processes of blocked actions
4    repeat
5       $AIR' := AIR$ ;  $APW' := APW$ 
6      for every process  $p$ 
7        for every local transition  $s_p \xrightarrow{b} s'_p$  scheduled by  $(B, AIR, APW)$  in  $s$ 
8          for every  $(IR, NR, PW) \in sigs(s_p, b)$ 
9            if  $APW \cup val(s) \models IR$  and  $APW \models NR$ 
10               $AIR := AIR \cup (IR \cap val(s))$ 
11               $APW := APW \cup PW$ 
12    until  $AIR = AIR'$  and  $APW = APW'$ 
13    return ( $FinalTest(s, AIR, APW)$ )
14
15  function FinalTest( $s, AIR, APW$ )
16    for every  $b$  enabled in  $s$  with  $proc(b) \notin proc(AIR) \cup proc(APW)$ 
17      if  $b$  is a write or a release action then return (false)
18      else if  $b = a_p(x, I)$  and  $APW(x) \subseteq I$  then return (false)
19      else if  $b = acq(\ell)$  and  $1 \notin APW(\ell)$  then return (false)
20    return (true)

```

6 Implementation and experimental results

We describe our implementation of the algorithm from Listing 1.1, and report on experiments on standard models from the literature. We compare six algorithms.

- Our two new algorithms: **pifs** the algorithm presented Listing 1.1, and **opifs** a variant storing only *special* nodes as described below.

Models	opifs			pifs			p-sets			spin			ic3	bdd
	N	time		N	Tr	time	N	time		N	time		time	time
bk, 3	1.4 k	0.0		3.6 k	557.0	0.0	10.7 k	0.0		11.7 k	0.0		TO	TO
bk, 4	86.4 k	15.8		197.1 k	1.6e+05	26.5	1.7 M	28.4		1.7 M	1.3		TO	TO
bk-lt, 3	331	0.0		912	85.0	0.0	2.7 k	0.0		3.0 k	0.0		272.0	637.4
bk-lt, 4	6.2 k	0.3		16.4 k	2.6e+03	0.4	129.0 k	1.4		118.6 k	0.1		TO	TO
bk-lt, 5	159.1 k	50.1		406.5 k	1.5e+05	91.8	9.2 M	171.6		7.0 M	7.2		TO	TO
bk-lt, 6	TO			TO			TO			573.6 M	1350.0		TO	TO
bk-b, 3	632	0.0		1.7 k	281.0	0.0	4.8 k	0.0		4.6 k	0.0		8.9	288.4
bk-b, 4	17.9 k	1.1		49.2 k	2.4e+04	1.6	357.3 k	3.8		284.2 k	0.4		TO	TO
bk-b, 5	650.2 k	305.4		1.8 M	5.3e+06	596.6	42.9 M	945.9		27.3 M	86.9		TO	TO
bk-ba, 3	277	0.0		803	95.0	0.0	2.8 k	0.0		3.0 k	0.0		9.3	710.3
bk-ba, 4	4.2 k	0.2		11.4 k	3.2e+03	0.3	152.7 k	1.6		141.2 k	0.1		TO	TO
bk-ba, 5	74.2 k	28.7		197.8 k	2.1e+05	58.2	14.4 M	271.0		11.2 M	23.9		TO	TO
dp-o, 20	8.1 k	0.9		29.1 k	1e+06	1.1	TO			TO			4.4	TO
dp-o, 29	33.0 k	6.9		119.6 k	5.4e+08	7.8	TO			TO			9.8	TO
dp-o, 30	37.9 k	7.9		135.8 k	1.1e+09	8.9	TO			TO			TO	TO
dp-o, 50	276.2 k	149.2		986.2 k	1.1e+15	154.2	TO			TO			TO	TO
dp-b, 11	256.6 k	20.9		724.1 k	4.8e+06	25.2	TO			TO			10.6	TO
dp-b, 13	1.3 M	146.4		3.6 M	7.7e+07	170.8	TO			TO			6.6	TO
dp-b, 15	6.2 M	913.6		17.4 M	1.2e+09	1151.3	TO			TO			8.3	TO
dp-ao, 16	33.9 k	3.9		86.5 k	1.7e+07	4.0	TO			TO			TO	TO
dp-ao, 30	473.5 k	149.5		1.1 M	3.6e+13	139.4	TO			TO			TO	TO
lz, 12	16.4 k	0.5		36.8 k	1.5e+04	0.7	11.8 M	298.6		16.1 M	41.1		23.0	TO
lz, 13	32.7 k	1.1		73.7 k	3.3e+04	1.6	43.9 M	1926.3		59.9 M	195.0		200.8	TO
lz, 14	65.5 k	2.5		147.4 k	7e+04	3.5	TO			223.6 M	884.0		149.9	TO
lz, 16	262.1 k	11.1		589.7 k	3.1e+05	16.2	TO			TO			TO	TO
lz, 18	1.0 M	50.4		2.4 M	1.4e+06	72.7	TO			TO			320.6	TO
lz, 20	4.2 M	230.8		9.4 M	6e+06	368.5	TO			TO			TO	TO
lz, 22	16.8 M	1330.8		37.7 M	2.6e+07	1948.0	TO			TO			TO	TO
pet, 3	41	0.0		124	32.0	0.0	210	0.0		276	0.0		13.5	7.6
pet, 4	630	0.0		1.7 k	2.8e+03	0.0	3.4 k	0.0		4.5 k	0.0		TO	TO
pet, 5	15.5 k	1.2		39.3 k	3.9e+06	1.3	106.1 k	1.4		129.6 k	0.2		TO	TO
pet, 6	633.6 k	150.2		1.5 M	2e+11	182.6	5.4 M	107.2		5.8 M	22.2		TO	TO
pet, 7	TO			TO			TO			371.8 M	1090.0		TO	TO
rcu, 7	26.4 k	2.1		93.1 k	7.8e+04	2.9	76.1 M	2115.2		55.4 M	147.0		2.9	2307.1
rcu, 8	100.8 k	9.7		355.7 k	3.9e+05	14.1	TO			602.5 M	2090.0		3.7	TO
rcu, 9	388.1 k	48.7		1.4 M	2e+06	74.0	TO			TO			13.5	TO
szy, 4	3.1 k	0.1		9.0 k	1.1e+05	0.1	34.5 k	0.2		29.1 k	0.0		1605.8	323.3
szy, 5	41.4 k	2.6		112.1 k	1.3e+08	3.3	868.5 k	9.0		603.1 k	1.1		TO	TO
szy, 6	620.9 k	69.0		1.6 M	1.6e+12	93.8	25.1 M	416.1		14.1 M	42.8		TO	TO
szy, 7	TO			TO			TO			365.9 M	762.0		TO	TO
tlb, 2, 1	125	0.0		552	24.0	0.0	12.9 k	0.1		14.1 k	0.0		TO	TO
tlb, 2, 2	607	0.0		2.6 k	160.0	0.0	307.4 k	4.3		327.9 k	0.9		TO	TO
tlb, 3, 1	13.6 k	1.1		58.0 k	2.8e+04	1.5	2.7 M	45.9		1.7 M	5.7		TO	TO
tlb, 3, 2	173.6 k	19.8		745.5 k	1.8e+06	25.8	TO			52.0 M	177.0		TO	TO
tlb, 4, 1	2.3 M	330.4		9.6 M	3e+08	425.8	TO			183.2 M	618.0		TO	TO
token, 3	990	0.0		2.0 k	3.9e+04	0.0	20.6 k	0.1		21.1 k	0.0		16.6	250.5
token, 4	31.9 k	1.3		65.6 k	1.7e+15	1.5	1.5 M	18.0		1.5 M	1.9		TO	TO
token, 5	589.6 k	34.0		1.2 M	1.9e+18	39.0	64.5 M	1562.7		62.9 M	150.0		TO	TO
token, 6	8.1 M	700.3		17.1 M	1.6e+19	806.5	TO			TO			TO	TO

Table 1. Comparison of our POR algorithms **pifs** and **opifs** with persistent sets technique (**p-sets**), and model-checkers **Spin** and **nuXmv**. N is the number of nodes in the constructed transition system, Tr is the number of full runs the transition system, $time$ is the running time in seconds, with a timeout of 40 min.

- **p-sets** is our implementation of *persistent sets* [21].
- **spin** corresponds to running the **Spin** model-checker [24] without any property to check.
- **nuXmv** tool [9] with two methods **smv-bdd** and **smv-ic3** corresponding to **check_ltl.spec** and **check_ltl.spec.ic3**, respectively. We test for unavoidability of the final state.

The **opifs** variant stores less nodes. It turns out that in transition systems constructed with our algorithm there are many nodes with only one outgoing transition. So we were tempted to exploit this situation. A node is *transitory* if it has one outgoing transition and the incoming transition is on the same process as the outgoing transition. The heuristic is to store only non-transitory nodes, and check the covering only with respect to non-transitory nodes. This increases computation time, but decreases the number of stored nodes. In our experiments

we evaluate this heuristic along the basic algorithm, as it is fast and sometimes gives significant reduction in the number of stored nodes.

We have spent a considerable effort to find meaningful models for our experiments. To compare different algorithms, models need to be parametric. We have considered various examples from the literature: **bk**, **bk-1t** and **bk-ba** are three variants of Lamport’s bakery algorithm [33]. **dp-o**, **dp-b** and **dp-so** are three versions of the well-known Dining Philosophers problem. The last zero example **1z** was introduced in [1]. **pet** is the Peterson’s mutual exclusion protocol [37]. Read-Copy-Update **rcu** is a protocol to ensure safe access to mutable shared data structures such as a linked list [14]. **szy** is a model of Szymanski’s mutual exclusion protocol [41]. The **t1b** shutdown protocol has been introduced in [7]. Finally, demand-driven token ring **token** is an example from [45]. Since we require models to be acyclic, each process performs a single round. For example, in a mutual exclusion algorithm, each process accesses the critical section once.

Table 1 presents the results of our experiments on the models described above. We have used a machine with Intel Xeon Skylake Gold 6240, 2.6 GHz. For the algorithms that build graphs we report the number N of nodes in the graph. Each model is described in a common format from which our scripts automatically produce models in the input language of each of the tools. This ensures that all the tools are run from the same description of the model. We ran the tools on several instances of each model, with a growing number of concurrent processes, indicated along the model name in Table 1. Model **t1b** has two parameters: the number of writers and the number of readers.

Comparison of stateful POR algorithms. Let us first focus on the four algorithms **opifs**, **pifs**, **p-sets** and **spin** which are based on stateful partial-order reduction techniques. Persistent sets and **spin** are current state-of-the-art stateful POR techniques. However, the partial-order technique implemented in **spin** does not provide any reduction on our models. Our models use shared variables and do not have local actions. It turns out that the POR technique in **spin** is meant towards optimizing concurrency between local actions, and some restricted form of asynchronous communications. We could not produce Promela programs matching these restrictions for our models. Hence, the number of nodes reported for **spin** correspond to the size of $TS(\mathbb{P})$, namely the whole state-space. On the other hand, **spin** runs much faster than our tool. This is due to our DFS engine as witnessed by the results from **p-sets** method where the additional computation in each node due to POR is very light.

Our implementation of persistent set technique **p-sets** follows [16]. The technique is not effective on our models due to state replication caused by sleep sets and a significant number of sleep-blocked runs. Only on some models like **pet** or **1z**, it produces significantly smaller graphs than the reachability graph (**spin**).

PIFS-based algorithms, **opifs** and **pifs**, compare very favorably. The gains in number of nodes N for algorithm **opifs** range from $9\times$ on **pet**, $22\times$ on **szy**, $40\times$ in other models, to $100\times$ on **token** and **t1b**. On model **rcu**, the gain is as high as $6000\times$. Running times per node are much slower than for **spin**, once again in part due to our DFS engine.

Comparison to symbolic techniques. The tool **nuXmv** implements various state-of-the-art symbolic verification algorithms. We have selected **ic3** and **bdd**, that provided the best results on our models. On most models, **nuXmv** could not construct the BDDs encoding the transition relation within 40 minutes. The **ic3** algorithm shows better performances on **dp-b** and **rcu**, but it does not scale up on all other models. This came as a surprise for us, since our models are acyclic (except for the self-loop on final states for **nuXmv** models).

We have also compared our approach to the specialized symbolic encodings implemented in **Dartagnan** [19]. Although **Dartagnan** and our tool do not work on the same inputs, we tried our best to produce C programs which are as close as possible to our models. We considered three models: Szymanski's algorithm (**szy**), Peterson's algorithm (**pet**) and **lz** under sequential consistency. **Dartagnan** performs much better on **lz** as it concludes within 9.8s for 22 processes. On **szy**, **Dartagnan** can check the same instances as our tool, but with significantly longer running time: 8.2s for 4 processes, then 38.2s for 5 processes, and 182s for 6 processes. It ran out of memory for 7 processes. Finally, our technique outperforms **Dartagnan** on **pet**, that runs out of memory for the instance with only 4 processes.

Although symbolic techniques are highly valuable, enumerative POR algorithms seem better suited for the verification of complex concurrent programs.

Comparison to stateless POR techniques. We compared our PIFS-based algorithms with the state-of-the-art tool **GenMC** [29]. As a benchmark, we used the same C programs as above for Szymanski's algorithm (**szy**), Peterson's algorithm (**pet**) and **lz**, under sequential consistency. Our experiments show that **GenMC** is able to verify Szymanski's algorithm with 2 processes (46 explored traces in 0.05s) and 3 processes (38471 traces in 3.15s). But it could not terminate within 40 minutes on the program with 4 processes which has an estimated number of 10^9 traces. The number of traces does not correspond to the number we report in Table 1 as neither **GenMC** nor our method are trace optimal for systems with await operation. The results for Peterson's algorithm are similar to those for Szymanski's algorithm. The results for **lz** are roughly equal for our methods and **GenMC**, although it is important to note that reduced transition system for this model is a tree, so stateful techniques lose their main advantage. **GenMC** could verify the model with 21 writers in 553s (about 10^7 traces, that this time is optimal), then 22 writers in 26m (roughly 2×10^7 traces), but it could not terminate within 40m for 23 writers. We believe that stateless techniques are very powerful to verify programs with complex operations, e.g. dynamic memory allocation, and few processes, for which storing the state-space is out of reach. Conversely, when intricate concurrent algorithms are considered, such as mutual exclusion algorithms, stateful techniques seem to be the preferred choice.

7 Conclusions

In this work, we derived a generic partial-order reduction method from the first principle of a strong diamond. This generic method, however, is computationally

expensive due to its reliance on *IFS*, an NP-complete oracle test. We therefore proposed an efficient heuristic, called *PIFS*, that approximates *IFS* and yields a correct and practical partial-order reduction algorithm. Our experimental evaluation demonstrates that this method is a valuable addition to the toolkit for analyzing concurrent pseudocode, capable of handling bigger models and producing substantially smaller reduced systems than existing approaches.

Stateful partial-order methods have received surprisingly little attention in the last decade, despite their potential. As our work demonstrates, there remains significant room for improvement in this area. Our *PIFS*-based approach, in particular, highlights the co-inductive facet of partial-order reduction. The role of the heuristic is to certify the non-existence of runs from a given state that satisfy certain properties. To do so more efficiently than by exhaustive search, *PIFS* constructs simple invariants.

As hinted in Section 4, the contextual independence relation at the core of our method generalizes naturally to other operations and even to arbitrary atomic blocks of code. This is an important consideration, as partial-order methods have been adapted to various frameworks, including actor programs [42], event-driven programs [2,25,34], and MPI programs [38]. While adapting the generic oracle to these settings appears straightforward, developing effective, syntax-aware heuristics like *PIFS* remains a non-trivial challenge.

Handling cycles in the state space, and thus removing our current acyclicity assumption, is another clear direction for future work. Existing solutions are often complex and have not been a major focus in recent literature, partly because the problem is orthogonal to the core challenge of finding optimal reductions. Nevertheless, addressing this limitation is essential for moving beyond bounded exploration to full program verification, even it is verification of a pseudocode.

Tackling these challenges—generalizing the heuristic and handling cycles—is a prerequisite for applying these methods to high-level specification languages like P and TLA+. This would be particularly interesting, as we believe that insights provided by partial-order reduction in this context go far beyond simple reachability checking.

Finally, extending our approach to weak memory models is another natural research direction. The notion of a global state is more nuanced in these models. Our method operates not on states alone, but on pairs of (state, sleep set), where the state component captures variable values and program counters. A first step would therefore be to adapt the sleep set component to correctly account for the behaviors allowed by weak memory.

8 Data-Availability Statement

The experimental results presented in Section 6 “Implementation and experimental results” can be reproduced from the artifact available at <https://doi.org/10.5281/zenodo.18017654>.

References

1. Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction. *Journal of the ACM*, 64(4):1–49, 2017.
2. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Frederik Meyer Bønneland, Sarbojit Das, Bengt Jonsson, Magnus Lang, and Konstantinos Sagonas. Tailoring Stateless Model Checking for Event-Driven Multi-threaded Programs. In Étienne André and Jun Sun, editors, *Automated Technology for Verification and Analysis*, pages 176–198. Springer Nature Switzerland, 2023.
3. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Sarbojit Das, Bengt Jonsson, and Konstantinos Sagonas. Parsimonious optimal dynamic partial order reduction. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part II*, volume 14682 of *Lecture Notes in Computer Science*, pages 19–43. Springer, 2024.
4. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lang, Tuan Phong Ngo, and Konstantinos Sagonas. Optimal stateless model checking for reads-from equivalence under sequential consistency. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.
5. Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. Stateless Model Checking Under a Reads-Value-From Equivalence. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, volume 12759, pages 341–366. Springer International Publishing, 2021.
6. Elvira Albert, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, Miguel Isabel, and Peter Stuckey. Optimal dynamic partial order reduction with context-sensitive independence and observers. *Journal of Systems and Software*, 202:111730, 2023.
7. D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation lookaside buffer consistency: A software approach. *SIGARCH Comput. Archit. News*, 17(2):113–122, 1989.
8. Marc Brooker and Ankush Desai. Systems Correctness Practices at Amazon Web Services. *Communications of the ACM*, 2025.
9. Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In Armin Biere and Roderick Bloem, editors, *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.
10. Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Data-centric dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2018.
11. Berk Cirisci, Constantin Enea, Azadeh Farzan, and Suha Orhun Mutluergil. A Pragmatic Approach to Stateful Partial Order Reduction. In Cezara Dragoi, Michael Emmi, and Jingbo Wang, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 13881, pages 129–154. Springer Nature Switzerland, 2023.
12. Camille Coti, Laure Petrucci, César Rodríguez, and Marcelo Sousa. Quasi-optimal partial order reduction. *Formal Methods in System Design*, 57(1):3–33, 2021.
13. Frank S. De Boer, Marcello Bonsangue, Einar Broch Johnsen, Violet Ka I Pun, S. Lizeth Tapia Tarifa, and Lars Tveito. SymPaths: Symbolic Execution Meets Partial Order Reduction. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel,

- Reiner Hähnle, and Mattias Ulbrich, editors, *Deductive Software Verification: Future Perspectives*, volume 12345, pages 313–338. Springer International Publishing, 2020.
14. Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Trans. Parallel Distributed Syst.*, 23(2):375–382, 2012.
 15. Azadeh Farzan. Commutativity in Automated Verification. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–7. IEEE, 2023.
 16. Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. Sound sequentialization for concurrent program verification. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 506–521. ACM, 2022.
 17. Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. Stratified Commutativity in Verification Algorithms for Concurrent Programs. *Proceedings of the ACM on Programming Languages*, 7(POPL):1426–1453, 2023.
 18. Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL’05*, 2005.
 19. Natalia Gavrilenko, Hernán Ponce-de-León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, volume 11561, pages 355–365. Springer International Publishing, 2019.
 20. Patrice Godefroid. Using partial orders to improve automatic verification methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer-Aided Verification*, pages 176–185. Springer, 1991.
 21. Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems An Approach to the State-Explosion Problem*. PhD thesis, Université de Liège, 1994.
 22. R. Govind, Frédéric Herbreteau, Srivathsan, and Igor Walukiewicz. Abstractions for the local-time semantics of timed automata: A foundation for partial-order methods. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–14. ACM, 2022.
 23. Frédéric Herbreteau, Sarah Larroze-Jardiné, and Igor Walukiewicz. Partial-order reduction is hard. In Patricia Bouyer and Jaco van de Pol, editors, *36th International Conference on Concurrency Theory, CONCUR 2025, August 26-29, 2025, Aarhus, Denmark*, volume 348 of *LIPIcs*, pages 22:1–22:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.
 24. Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
 25. Casper S. Jensen, Anders Moller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. Stateless model checking of event-driven applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015*, pages 57–73. Association for Computing Machinery, 2015.
 26. Bengt Jonsson, Magnus Lang, and Konstantinos Sagonas. Awaiting for Godot Stateless Model Checking that Avoids Executions where Nothing Happens. In *{22nd Formal Methods in Computer-Aided Design, {FMCAD} 2022}*. IEEE, 2022.
 27. Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643, pages 398–413. Springer Berlin Heidelberg, 2009.

28. Shmuel Katz and Doron Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2):107–120, 1992.
29. Michalis Kokologiannakis, Rupak Majumdar, and Viktor Vafeiadis. Enhancing GenMC’s Usability and Performance. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 66–84. Springer Nature Switzerland, 2024.
30. Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Truly stateless, optimal dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–28, 2022.
31. Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. Unblocking Dynamic Partial Order Reduction. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, volume 13964, pages 230–250. Springer Nature Switzerland, 2023.
32. Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22–26, 2019*, pages 96–110. ACM, 2019.
33. Leslie Lamport. A new solution of dijkstra’s concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.
34. Pallavi Maiya, Rahul Gupta, Aditya Kanade, and Rupak Majumdar. Partial Order Reduction for Event-Driven Multi-threaded Programs. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 680–697. Springer, 2016.
35. Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. Trace aware random testing for distributed systems. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.
36. Doron Peled. Partial-order reduction. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 173–190. Springer, 2018.
37. Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.
38. The Anh Pham. *Efficient State-Space Exploration for Asynchronous Distributed Programs: Adapting Unfolding-Based Dynamic Partial Order Reduction to MPI Programs*. These de doctorat, Rennes, Ecole normale supérieure, 2019.
39. Maxwell Pirtle, Luka Jovanovic, and Gene Cooperman. McMini: A Programmable DPOR-Based Model Checker for Multithreaded Programs, 2023.
40. César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based Partial Order Reduction. *LIPIcs, Volume 42, CONCUR 2015*, 42:456–469, 2015.
41. Boleslaw K. Szymanski. A simple solution to lamport’s concurrent programming problem with linear wait. In Jacques Lenfant, editor, *Proceedings of the 2nd international conference on Supercomputing, ICS 1988, Saint Malo, France, July 4–8, 1988*, pages 621–626. ACM, 1988.
42. Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems*, pages 219–234. Springer, 2012.
43. Rahmadi Trimananda, Weiyu Luo, Brian Demsky, and Guoqing Harry Xu. Stateful Dynamic Partial Order Reduction for Model Checking Event-Driven Applications that Do Not Terminate, 2021.

44. Antti Valmari. Stubborn sets for reduced state space generation. In G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, and Grzegorz Rozenberg, editors, *Advances in Petri Nets 1990*, volume 483, pages 491–515. Springer Berlin Heidelberg, 1991.
45. Antti Valmari. A state space tool for concurrent system models expressed in C++. In Jyrki Nummenmaa, Outi Sievi-Korte, and Erkki Mäkinen, editors, *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST'15), Tampere, Finland, October 9-10, 2015*, volume 1525 of *CEUR Workshop Proceedings*, pages 91–105. CEUR-WS.org, 2015.
46. Antti Valmari and Henri Hansen. Stubborn Set Intuition Explained. In Maciej Koutny, Jetty Kleijn, and Wojciech Penczek, editors, *Transactions on Petri Nets and Other Models of Concurrency XII*, volume 10470, pages 140–165. Springer Berlin Heidelberg, 2017.
47. Bjorn Wachter, Daniel Kroening, and Joel Ouaknine. Verifying multi-threaded software with impact. In *2013 Formal Methods in Computer-Aided Design*, pages 210–217. IEEE, 2013.
48. Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole Partial Order Reduction. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963, pages 382–396. Springer Berlin Heidelberg, 2008.
49. Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. In Klaus Havelund, Rupak Majumdar, and Jens Palsberg, editors, *Model Checking Software*, pages 288–305. Springer, 2008.

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Deconstructing Subset Construction

Reducing While Determinizing

John Nicol¹ and Markus Frohme²

¹ Unaffiliated, United States
jnicol1994@alumni.cmu.edu

² TU Dortmund University, Germany
markus.frohme@tu-dortmund.de

Abstract. We present a novel perspective on the NFA canonization problem, which introduces intermediate minimization steps to reduce the exploration space on-the-fly. Central to our approach are *equivalence registries* which track and unify language-equivalent states, and allow for additional optimizations such as convexity closures and simulation. Due to the generality of our approach, these concepts can be embedded in classic subset construction or Brzozowski's approach. We evaluate our approach on a set of synthetic and real-world examples from automatic sequences and observe that we are able to improve especially worst-case scenarios. We provide an open-source library implementing our approach.

Keywords: Automata · Canonization · Minimization · Determinization

1 Introduction

Finite-state machines are foundational models of computation in computer science, where their simplicity and expressive power make them of both theoretical and practical interest. Among the most prominent types are *deterministic finite automata* (DFAs) and *nondeterministic finite automata* (NFAs), which are extensively studied for their applications in various domains. DFAs are often preferred over NFAs due to their efficiency in operations such as testing language equivalence and inclusion, universality, complementation, and other closure properties [21, 23], trace inclusion (noted in [20]), and string matching [2]. Many of these operations play a critical role in practical applications such as model-based testing [24, 8, 48] and theoretical fields such as automatic sequences [4, 36].

Rabin and Scott [31] have shown that DFAs and NFAs are equally expressive by transforming NFAs into language-equivalent DFAs via *subset* (or *powerset*) *construction*. The resulting DFAs may be of exponential size compared to the original NFAs and may not be minimal. The latter can be easily addressed via an additional minimization step [22] which is often required if the resulting DFAs need to be processed further. A minimal DFA is called *canonical* since a minimal DFA is unique up to isomorphism [23]. Constructing the canonical DFA for a given NFA is called the *canonization problem* [20, 46].

The exponential blowup of canonization can be a challenge in practice. However, many NFAs exhibit a notable *compression ratio*, i.e., the ratio between the size of the determinized (via subset construction) DFA and its minimized version. Real-world cases report compression ratios ranging from 30 ([15]), to 2500 ([44]), to 5 million ([34, 37], and addenda for [29]). In such situations, materializing the fully determinized automaton is often the bottleneck of the canonization process.

For certain NFA-related problems, there exists extensive research on limiting the impact of state blowup, e.g., by using *antichains* for testing universality and language inclusion [14, 51] or *bisimulation up to congruence* for testing language equivalence [6]. See [11, 18] for surveys on the current state of this work. For the canonization problem, however, the construction of the full DFA is necessary.

In this paper, we present OTF – an *on-the-fly* approach for canonizing NFAs. Instead of treating determinization and minimization as distinct phases of the canonization process, OTF interleaves these concepts by repeatedly minimizing the (partially) constructed DFA of subset construction. Insights about equivalent states of the DFA under construction can be used to detect equivalences even for yet unseen subsets of NFA states and are fed back to subset construction in order to reduce the state space requiring exploration. This form of communication is formalized via *equivalence registries* with which the process can be parameterized, giving OTF a framework character. We present two exemplary implementations of equivalence registries. In our evaluation, we show that OTF is able to improve the performance of canonization for complex real-world NFAs, while remaining competitive for simple(r) systems. We implement our approach in an open-source library [30] that is based on AutomataLib [3] and included in the theorem prover Walnut [50] since version 7.

A noteworthy alternative to classic NFA canonization is Brzozowski’s algorithm [9] which first determinizes (e.g., via subset construction) the reversed NFA, then reverses the constructed DFA, and finally determinizes it again. This method canonizes the original NFA without requiring any minimization steps and can outperform classic subset construction in some cases. Since it relies only on determinization, it can make use of OTF as well and we include it in the evaluation to show the flexibility of our approach. Finally, [20] similarly considers the potential of compression ratio by generalizing subset construction; we do a comparison against their best-performing algorithm.

Outline Section 2 introduces preliminary notation and concepts used throughout the paper. In Section 3, we discuss the general OTF framework for which Section 4 presents two exemplary equivalence registries. In Section 5, we evaluate the performance of OTF and compare our approach with the alternatives. Section 6 concludes the paper and gives an outlook on future research.

2 Preliminaries

This section introduces basic notations and concepts used throughout the paper.

2.1 Finite Automata

Let Σ denote a finite, non-empty set of input symbols and Σ^* denote the set of finite words over Σ with ε denoting the empty word.

Definition 1 (Finite Automaton). A finite automaton (FA) is a tuple $A = (S, \Sigma, \delta, S_0, F)$ such that

- S denotes a finite, non-empty set of states,
- Σ denotes a finite, non-empty set of input symbols,
- $\delta \subseteq S \times \Sigma \times S$ denotes a labeled transition relation,
- $S_0 \subseteq S$ denotes a set of initial states, and
- $F \subseteq S$ denotes a set of final (or accepting) states.

Let 2^X denote the set of subsets of a set X . By abuse of notation, we may interpret δ as a function $S \times \Sigma \mapsto 2^S$ such that $\delta(s, i) = \{t \in S \mid (s, i, t) \in \delta\}$. We generalize this view to sets of states and words such that $\delta(Q, i) = \bigcup_{q \in Q} \delta(q, i)$ and $\delta(s, \varepsilon) = s, \delta(s, iw) = \delta(\delta(s, i), w)$ for $Q \subseteq S, s \in S, i \in \Sigma, w \in \Sigma^*$. In the following, we may refer to a set of states Q as a *metastate*.

A word w is *accepted* by a state $s \in S$ iff $\delta(s, w) \cap F \neq \emptyset$. The language of a state $s \in S$, $L(s)$, is defined as set of accepted words of s . The language of a metastate Q is defined as $L(Q) = \bigcup_{q \in Q} L(q)$. The language of an FA A is defined as $L(A) = L(S_0)$. Two FAs, A and B , are said to be *language-equivalent* iff $L(A) = L(B)$. We call A a *deterministic* finite automaton (DFA) iff $|S_0| = 1$ and $|\delta(s, i)| \leq 1$ for all $s \in S, i \in \Sigma$. Otherwise, A may be referred to as a *non-deterministic* finite automaton (NFA).

2.2 Simulation and Bisimulation

For reasoning about the behavioral relationships between states of an FA, there exist various notions of equivalence. A popular one is (bi-) simulation [28].

Definition 2 (Simulation). Let A be an FA. A binary relation $R \subseteq S \times S$ is called a simulation relation for A iff $(x, y) \in R$ implies

- $x \in F \Rightarrow y \in F$, and
- $\forall i \in \Sigma: (x, i, x') \in \delta \Rightarrow (\exists y' \in S: (y, i, y') \in \delta \wedge (x', y') \in R)$.

Definition 3 (Bisimulation). Let A be an FA. A binary relation $R \subseteq S \times S$ is called a bisimulation relation for A iff R and its inverse R^{-1} are simulation relations for A .

Note that bisimulation is stricter than simulation equivalence, as it requires the inverse of the *same* relation to also be a simulation.

The coarsest simulation and bisimulation relations, known as *similarity* and *bisimilarity*, can be computed in $\mathcal{O}(|\delta| \cdot |S|)$ time [32] and $\mathcal{O}(|\delta| \cdot \log |S|)$ time [49], respectively. It is well known that simulation implies finite trace (i.e., language) inclusion [12, 19, 20] and bisimulation implies finite trace (i.e., language) equivalence [19]. Thus, bisimulation or simulation equivalences can be used to quotient NFAs, which is often done to reduce the size of NFAs before determinizing. Note that this step is only relevant for nondeterministic systems since for DFAs, minimization and bisimulation quotienting coincide [6].

2.3 Semilattice, Convexity, Antichain

Let X denote a finite set. A *semilattice* $(X, \circ)$ is an algebraic structure that consists of X and a binary operation $\circ : X \times X \rightarrow X$ which is associative, commutative, and idempotent. Let $(X, \circ)$ be a semilattice and $M \subseteq X$. We call $(M, \circ)$ a (semi-) *sub-lattice* of $(X, \circ)$ iff $(M, \circ)$ is a well-defined semilattice itself. We call a semilattice $(M, \circ)$ *convex* iff $\{y \in X \mid \exists x, z \in M, x \preceq y \preceq z\} \subseteq M$. Here, $\preceq$ denotes the preorder related to the respective (join/meet) operator $\circ$. In this paper, we mainly consider variations of the well-known powerset lattice $(2^X, \cup, \cap)$. In particular, $(2^X, \cup)$ is a join-semilattice.

Let $L = (M, \circ)$ be a semilattice. The *convexity closure* of L is obtained by adding all missing elements to L in order to make it convex. For example, let $X = \{1, \dots, 5\}$, $A = \{1, 2\}$, $B = \{3, 4\}$. Then $L_1 = (\{A, B, A \cup B\}, \cup)$ is a (semi-) sub-lattice of $(2^X, \cup)$ and the convexity closure of L_1 is given by

$$(\{\{1, 2\}, \{3, 4\}, \{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, \{1, 2, 3, 4\}\}, \cup).$$

In particular, the convexity closure of L_1 is a well-defined sublattice of $(2^X, \cup)$.

Testing whether an element is contained in a convex (join-) semilattice L can be efficiently implemented by testing whether it is covered from above by L 's greatest element and covered from below by any of L 's minimal elements. Let $L = (Y, \cup)$ denote a (join-) semilattice for some $Y \subseteq 2^X$. The *greatest element* of L is given by $\bigcup L = \bigcup_{Y_i \in Y} Y_i$. An element $Y_i \in Y$ is called *minimal* iff $\nexists Y_j \in Y : Y_j \subset Y_i$. Here, $\subset$ is the related preorder of the join operator $(\cup)$ without equality.

For L_1 , the greatest element is $\{1, 2, 3, 4\}$ and the minimal elements are $\{1, 2\}$ and $\{3, 4\}$. The element $\{1, 2, 3\}$ is contained in the convexity closure of L_1 because $\{1, 2\} \subseteq \{1, 2, 3\}$ and $\{1, 2, 3\} \subseteq \{1, 2, 3, 4\}$. It is worth noting that the minimal elements of L form an antichain, i.e., a set of pairwise incomparable (in the sense of $\preceq$) elements, and can be efficiently stored in a list-like structure [10].

3 The OTF Framework

This section presents how we integrate the OTF concept into the determinization process. Intuitively, our approach is similar to classic subset construction with two major modifications.

First, the **NFA** state space exploration may be repeatedly interrupted by a *threshold* predicate. During this interrupt, the partially constructed **DFA** is minimized by an efficient algorithm such as [22]. On the one hand, this reduces the size of the intermediate **DFA** model. On the other hand, the information about equivalent **DFA** states may be used to optimize the remaining exploration of the **NFA** state space (see below).

Second, the mapping between **NFA** metastates and **DFA** states, that is established during subset construction is handled by an *equivalence registry*. Equivalence registries allow one to incorporate *external* information into this mapping either by means of preprocessing steps such as (bi-)simulation relations on **NFA**

states or in-processing steps such as [DFA](#) state equivalences arising from minimization. For this, they need to support a specific set of operations.

Definition 4 (Equivalence Registry). *Let $A = (S, \Sigma, \delta, S_0, F)$ be the input [NFA](#) of the determinization process and $B = (S', \Sigma, \delta', S'_0, F')$ be the resulting output [DFA](#) with $L(A) = L(B)$. Furthermore, let $Q \subseteq S$ be a metastate of A and $q, q_1, q_2 \in S'$ be states of B . An equivalence registry is a data structure that supports the following operations:*

- *GET(Q) returns a state $q \in S'$ such that $L(Q) = L(q)$. The method may also return undefined if it is not aware of such state yet.*
- *PUT(Q, q) links the behavior of metastate Q to state q with the precondition that $L(Q) = L(q)$.*
- *UNIFY(q_1, q_2) informs the registry about the language equivalence of states q_1 and q_2 , which may affect future results of the GET method.*

The goal of equivalence registries is to allow for potentially more involved lookups of mapped states in order to reduce the number of explored [NFA](#) metastates and save runtime. For example, to replicate classic subset construction, one can implement a one-to-one equivalence registry with a hash-based data structure to run GET and PUT in (amortized) constant time and implement UNIFY as a no-op. However, this also results in the full exploration of the [NFA](#) state space. Relaxing the GET operation and allowing one to look up mapped states *modulo language equivalence* has huge potential for reducing the metastate space that needs to be explored, thus improving overall determinization performance. [Section 4](#) looks at potential registry implementations in more detail.

We design OTF as a *framework* by allowing the threshold and equivalence registry to be parameters to this process. In practice, this allows users to construct highly customizable canonization processes tailored to their use cases. We continue with outlining some of the overarching structure of our approach.

3.1 Algorithmic Sketch

[Algorithm 1](#) formalizes and outlines a possible implementation. While our full implementation includes additional technical optimizations such as batching calls or re-using buffers, we focus here on the general concepts.

The algorithm starts by initializing a counter used to identify states of the resulting [DFA](#) and creates the corresponding state sets (regular, initial, and final). The equivalence registry and exploration queue³ Q are then initialized with information about the initial [NFA](#) metastates. The set E' tracks the fully explored states of the final [DFA](#) A' .

The main exploration loop starts at [Line 12](#). The *current* metastate C is removed from the queue and its representative c' is retrieved from the registry R .

³ We choose a last-in-first-out queue (stack) which traverses the [NFA](#) state space depth-first. In our experiments, this has yielded better results as cycles are found earlier, improving the detection of equivalent states.

Algorithm 1 On-The-Fly Determinization**Input:** NFA $A = (S, \Sigma, \delta, S_0, F)$, predicate THRESHOLD, registry R **Output:** A language-equivalent DFA $A' = (S', \Sigma, \delta', S'_0, F')$

```

1:  $cntr \leftarrow 0$  ▷ Initialization
2:  $S' \leftarrow \{cntr\}$ 
3:  $S'_0 \leftarrow \{cntr\}$ 
4: if  $S_0 \cap F \neq \emptyset$  then
5:    $F' \leftarrow \{cntr\}$ 
6: else
7:    $F' \leftarrow \emptyset$ 
8: end if
9:  $PUT_R(S_0, cntr)$ 
10:  $Q \leftarrow \{S_0\}$ 
11:  $E' \leftarrow \emptyset$ 
12: while  $Q \neq \emptyset$  do ▷ Exploration loop
13:    $C \leftarrow POP_Q()$ 
14:    $c' \leftarrow GET_R(C)$ 
15:   for  $i \in \Sigma$  do
16:      $N \leftarrow \delta(C, i)$ 
17:      $n' \leftarrow GET_R(N)$ 
18:     if  $n'$  is undefined then
19:        $cntr \leftarrow cntr + 1$ 
20:        $n' \leftarrow cntr$ 
21:        $S' \leftarrow S' \cup \{n'\}$ 
22:       if  $N \cap F \neq \emptyset$  then
23:          $F' \leftarrow F' \cup \{n'\}$ 
24:       end if
25:        $PUT_R(N, n')$ 
26:        $PUSH_Q(N)$ 
27:     end if
28:      $\delta'(c', i) \leftarrow n'$ 
29:   end for
30:    $E' \leftarrow E' \cup \{c'\}$ 
31:   if THRESHOLD( $A'$ ) then ▷ On-the-fly minimization
32:     for  $i \in \{0, \dots, cntr\}$  do
33:       if  $i \in F' \cap E'$  then
34:          $sig[i] = \text{true}$ 
35:       else if  $i \in S' \cap E'$  then
36:          $sig[i] = \text{false}$ 
37:       else
38:          $sig[i] = i$ 
39:       end if
40:     end for
41:     for  $(s', t') \in MINIMIZE(A', sig)$  do
42:        $UNIFY_R(s', t')$ 
43:     end for
44:   end if
45: end while
46: return  $A'$ 

```

Every element in Q has a representative in R , ensuring that c' is always defined. For each input symbol, the *next* metastate N is computed and its representative n' is looked up. If n' is not found, a new state is added to A' by incrementing the counter and updating the relevant variables. The registry is then updated and N is queued for future exploration. Afterward, the respective transition can be set. Once all outgoing transitions of c' are set, c' is marked as explored.

In [Line 31](#), the `THRESHOLD` predicate determines if minimization should occur. To do so, the predicate may analyze various aspects of A' including the number of states or transitions. The check can also be stateful in that it only passes if a certain increase (e.g., in size) from previous invocations is met. To perform the minimization, an associative array *sig* is built, storing each state's *signature* using a Boolean value for explored states' acceptance or a unique numerical identifier for unexplored states.

A' is then minimized using *sig* as the initial partition. Note that A' may be partial during the exploration loop but the minimization procedure may easily add an implicit sink state to ensure a correct result (e.g., when using Hopcroft's algorithm [\[22\]](#) for minimization). For convenience, we assume that `MINIMIZE` mutates A' directly, eliminating the need for explicit updates to related sets and functions. Additionally, `MINIMIZE` returns pairs of equivalent states which are forwarded to the registry to refine future lookups (cf. [Section 4](#)). The exploration loop then proceeds with the next metastate.

Once the exploration queue is empty, the final iteration of A' is returned. While A' may be minimized during exploration, the threshold predicate might not trigger in the last iteration. Thus, a final minimization is necessary for correctly canonizing A . This step is omitted from [Algorithm 1](#) to allow for utilizing the presented technique directly in Brzozowski's approach as well (cf. [Section 5](#)) which does not require an explicit minimization.

3.2 Termination and Correctness

With a threshold that constantly returns `false` and a registry that implements a one-to-one mapping between metastates and representatives, [Algorithm 1](#) coincides with classic subset construction. Thus, we omit general discussions about termination and correctness, as they follow directly from the original algorithm. Instead, we focus on how our modifications preserve these properties.

The first modification relaxes `GET` to return representatives of language-equivalent metastates instead of previously mapped ones. Once classic subset construction terminates, these metastates correspond to equivalent [DFA](#) states, which can be permuted or merged without affecting the language of the [DFA](#). Allowing these transformations earlier (during lookup) only shortcuts these steps while preserving language equivalence.

The second modification introduces intermediate minimizations, where unexplored states are assigned unique blocks in the initial partition, ensuring they remain distinct until their behavior is fully determined. Consequently, minimization never merges states with incomplete information. Only explored states with identical acceptance and identical successors are merged, meaning that the final

DFA remains equivalent to its non-minimized counterpart. Thus, intermediate minimizations do not alter the **DFA**'s language, maintaining parity with classic subset construction.

3.3 Complexity

Providing a detailed analysis of OTF's complexity is a challenge as it highly depends on user-provided parameters such as the costs of the GET/PUT/UNIFY implementations, how often the THRESHOLD predicate triggers, or the compression factor of the constructed **DFAs**. As a result, we only want to briefly sketch the worst-case performance of OTF and refer the reader to [Section 5](#) for practical performance results.

The dominating factor of **NFA** canonization is the state explosion problem of subset construction when determinizing the **NFA**. There exist **NFAs** A whose language-equivalent **DFA** representation requires an exponential (in the size of A) number of states [43]. As OTF also constructs a language-equivalent **DFA**, it may have to iterate its main loop (cf. [Line 12](#)) exponentially often. On top of that come the costs of registry management, threshold evaluation, and minimization, which – in the worst case – results in additional overhead compared to classic subset construction. [Section 4](#) provides more details on the complexity of the specific registries presented in this paper.

4 Equivalence Registries

The performance of OTF highly depends on the chosen equivalence registry. Intelligently looking up and effectively unifying language-equivalent metastates can drastically mitigate the state explosion problem of classic subset construction. In the following, we present two exemplary implementations: *CCL* generalizes state equivalences obtained from intermediate minimizations via convexity closures and *CCLS* further uses state-level preorders on the input **NFA** (e.g., similarity) to normalize metastates before lookup.

4.1 CCL – Convexity Closure Lattice

The use of convexity closures of lattices exploits a fundamental observation of Bonchi and Pous [6] that for two metastates A, B of an **NFA**, we have

$$L(A \cup B) = L(A) \cup L(B).$$

Exploiting this property further, we can conclude for $L(A) = L(B)$ that

$$L(A \cup B) = L(A) \cup L(B) = L(A) = L(B). \quad (1)$$

This relationship is not limited to disjoint metastates. Consider the situation where $L(A) = L(A \cup B)$. Here, no state in $B \setminus A$ impacts the language of either $L(A)$ or $L(A \cup B)$. In particular, for every X such that $A \subseteq X \subseteq A \cup B$, we have

$$L(A) = L(X) = L(A \cup B). \quad (2)$$

The same reasoning applies to the case of $L(B) = L(A \cup B)$, placing us in a scenario similar to [Section 2.3](#): detecting equivalence between two metastates allows one to construct a (join-) semilattice L with minimal elements A, B and greatest element $A \cup B$. L itself forms a sub-lattice for the powerset lattice of the global metastate space. Due to [Equations \(1\) and \(2\)](#), language equivalence holds for all elements in the convexity closure of L . Thus, recognizing equivalence between two metastates A, B provides insight into the behavior of many (potentially yet unobserved) metastates.

The CCL registry maintains a mapping between traversed states of the [DFA](#)-under-construction and lattices that cover their respective language-equivalent metastates. Our implementation realizes the respective operations as follows.

PUT(Q, q) Initially, when metastate Q is linked with state q , a singleton lattice is stored. Using a hash-based data structure, this can be implemented in (amortized) constant time.

UNIFY(q_1, q_2) When being informed of the equivalence of two states, their associated lattices L_1, L_2 must be merged. If L_1, L_2 are singletons, this process coincides with the example of [Section 2.3](#) and is implemented straightforwardly in linear time (for set union). If either lattice has undergone previous merges, we can apply an abstracted convexity closure: since all metastates covered by L_1, L_2 are language-equivalent, we select their greatest elements G_1, G_2 as single metastate representatives to apply the same steps as before. The greatest element of the new lattice L' is given by the joined metastate $G_1 \cup G_2$. Due to union being the join operator, this ensures that L' covers all elements of L_1, L_2 from above. The new minimal elements of L' are constructed from the union of minimal elements M_1, M_2 of L_1, L_2 , respectively. Here, elements in M_1, M_2 may be in a subset relation which would violate their minimality in L' . To tackle this issue, we filter out these elements which requires at most $\mathcal{O}(|M_1||M_2|)$ subset comparisons which are the dominating operations of this case.

GET(Q) In order to look up representatives, we first check whether Q is a singleton lattice via a hash-based lookup in (amortized) constant time. If a lattice is found, we return the single representative which trivially satisfies the required language equivalence properties. Otherwise, we iterate over the set of lattices to find one that covers Q from above and below. Recall from [Section 2.3](#) that the convexity closure of a lattice can be efficiently stored using its antichain of minimal elements and greatest element. Thus, the registry first scans for eligible lattices whose greatest element is a superset of Q . Then, for each candidate, the respective antichain is searched for elements that are a subset of Q . If such a candidate is found, the associated representative of the lattice is returned. By construction of the lattices and the coverage from above and below, the returned representative is guaranteed to be language-equivalent to Q . If no such lattice is found, we return *undefined*.

Remarks The data structures underlying CCL (a minimal antichain plus a greatest element, cf. [Section 2.3](#)) are reminiscent of antichain algorithms for universality and language inclusion [[14](#), [51](#)], where one maintains reachable metastates modulo one-sided subsumption to avoid redundant exploration (detecting non-universality/inclusion violations early). The role here is different: CCL uses antichains to summarize convexity-closed equivalence regions, driven by equalities obtained from intermediate minimization.

Our implementation uses simple list-based structures which in the worst-case require a quadratic (in the number of lattices and the number of their minimal elements) number of subset comparisons for the lookup. Future work may look at more involved structures for antichains [[10](#)] or inverse indices for faster subset checking [[25](#)].

4.2 CCLS – CCL with Similarity

The CCL registry is only able to exploit knowledge about equivalent states from calls to UNIFY. Thus, the performance of the CCL registry is inherently linked to the effects of minimization. The CCLS registry refines the CCL approach by introducing an *external* preprocessing step which computes the similarity relation on the original NFA ahead of the determinization process. This can potentially accelerate metastate equivalence detection *independently* of any minimizations.

To illustrate this concept, let a, b be NFA states and let a simulate b , i.e., $b \preceq a$. Then $L(b) \subseteq L(a)$ which implies $L(\{a, b\}) = L(\{a\}) \cup L(\{b\}) = L(\{a\})$. In metastate terms, we can replace $\{a, b\}$ with $\{a\}$ (pruning) and $\{a\}$ with $\{a, b\}$ (saturating) without affecting language equivalence. This principle extends to supersets as well, e.g., $\{a, b, c\}$ can be pruned to $\{a, c\}$ and $\{a, c\}$ can be saturated to $\{a, b, c\}$. We integrate this concept into the registry operations as follows.

PUT(Q, q) First, we prune Q into Q_p and saturate Q into Q_s , which requires linear time in the size of Q if appropriate caching of simulation relations is used. If $Q_p \neq Q_s$, instead of storing a singleton lattice like CCL, we directly create a multi-element lattice with Q_p as a single minimal element and Q_s as the greatest element. This lattice may already cover a larger metastate space *for free* and therefore reduce the search space of future lookups. By simulation and the results of [Section 4.1](#), all covered elements are language-equivalent to q .

UNIFY(q_1, q_2) Merging lattices is implemented identical to [Section 4.1](#).

GET(Q) For looking up representatives, we prune Q into Q_p and then search for a lattice covering Q_p similar to CCL. The goal of this transformation is to find a representative more often than CCL because Q_p may be smaller than Q and therefore may be covered by smaller lattices. Due to the language equivalence of Q and Q_p , the retrieved representative of Q_p is also a valid representative of Q , ensuring correctness of the result. While pruning introduces additional linear costs, the runtime is still dominated by the quadratic lookup process.

Remarks Simulation preorders have been used in several ways in the past. In antichain decision procedures for inclusion/universality [14, 51] and in particular in the combination of simulation and antichains [1], simulation is used to strengthen one-sided subsumption pruning (to normalize metastates by removing simulation-dominated states) in order to reduce exploration. In contrast, [12, 20] also employ simulation as a normalization/reduction device that can act in both directions by removing dominated structure and, in the saturation sense, adding implied structure, albeit for different objectives (automata reduction and determinization variants, respectively). CCLS adopts the preorder-based normalization in a different role: for each metastate Q we use pruning and saturation *simultaneously* as a two-sided summary, increasing PUT coverage and improving GET hit rates via the same convexity-membership criterion as in CCL.

Note that the effectiveness of the CCLS registry depends on the use case. Computing the simulation preorder can be prohibitively expensive for large or dense NFAs, so the benefit of precomputing simulation relations must outweigh the cost. On an implementation level, we preprocess the NFA's initial and final states following techniques from [12]. Additionally, since the CCLS registry requires similarity information at construction time, we compute it before running OTF and use it to further quotient the system via simulation equivalence. This also allows us to avoid challenges such as cyclic subset inclusions and tie-breaking during metastate selection [20].

5 Evaluation

This section compares the performance of OTF with classic NFA canonization algorithms. For the evaluation, we look at two different use cases covering practical examples from a real-world application and synthetically generated systems that exhibit specific structural properties. We continue with a brief overview of the general benchmark structure and discuss the two use cases afterward.

5.1 Setup

Table 1 summarizes the considered algorithms and configurations used in the evaluation. **OTF** describes our presented canonization process using the CCL equivalence registry, whereas **OTF-S** describes the variant using the CCLS equivalence registry. **SC** describes the canonization process via classic subset construction, whereas **SC-S** describes a variant with pruning and saturation. Internally, we use the OTF-S implementation with a threshold that constantly returns **false**, preventing intermediate minimizations. This closely resembles `SUBSET(compress⊆)` from [20]. **BRZ** describes Brzozowski's [9] canonization algorithm, whereas **BRZ-S** describes a variant using SC-S for the first determinization phase. Recall that BRZ constructs canonical automata without any minimization, i.e., the second phase does not create any equivalent states. As a result, we use plain SC for the second phase in this configuration. **BRZ-OTF(-S)** describes a variant of BRZ using OTF(-S) for the first phase.

Table 1. Algorithms and configurations used in the benchmarks.

		Uses intermediate minimizations?	
		No	Yes
Uses simulation?	No	SC BRZ	OTF OTF-BRZ
	Yes	SC-S BRZ-S	OTF-S OTF-BRZ-S

The benchmarks were executed on a server with two AMD EPYC 7763 CPUs with 2TB of RAM. The resources were allocated to run multiple benchmarks in parallel. Individual runs were single-threaded and limited to 256GB of RAM.

Parameters All configurations use Hopcroft’s algorithm [22] to minimize either the intermediate automata (OTF) or the final automaton (OTF, SC). OTF and BRZ-OTF configurations use an *adaptive* threshold which triggers a minimization after every t calls to the THRESHOLD predicate (cf. Line 31). t is initialized to 5000 and updated after every minimization according to the formula

$$t_{new} = t_{old} \cdot \frac{s_{new}}{s_{old}}$$

where s_{new} and s_{old} denote the number of states of the freshly and previously minimized DFA, respectively⁴. This means that if the constructed DFA does not compress well, i.e., $s_{new} > s_{old}$, the threshold and thus the minimization interval increases. In contrast, when observing effective reductions in size, i.e., $s_{new} < s_{old}$, the threshold triggers minimizations more often to adaptively exploit this trend. Furthermore, we limit the minimal value and the maximum increase of t to 5000 in order to better deal with extreme corner cases.

Preprocessing To normalize the input automata, we further apply some typical redundancy elimination techniques in a preprocessing step, trimming each automaton (i.e., removing non-accessible and non-co-accessible states) and quotienting according to the concerned configuration either by bisimulation [49] or simulation equivalence [12, 32]. We found the LIGHT- and HEAVY- reduction methods from [12] to be impractical in our benchmarks due to their worst-case performance on NFAs. However, we include their preprocessing steps of modifying initial and final states to find more simulation relations.

Measurements For our evaluation, we look at the total runtime of the canonization process (including preprocessing and final minimization), the *intermediate automaton overhead*, and the number of minimizations. The overhead describes the difference between the number of states of intermediate automata

⁴ We initialize s_{old} to 5000 to prevent a division-by-zero on the first invocation.

Table 2. Number of times that the adaptive threshold triggered a minimization for the Walnut examples.

	min	median	max	average
OTF	2	48	145	50.29
OTF-S	0	9	105	14.93
BRZ-OTF	0	2	27	4.61
BRZ-OTF-S	0	1	17	3.49

and the final [DFA](#). For SC, we use the size of the determinized automaton as reference point. For OTF and BRZ-OTF, we use the maximum size of the intermittently minimized automata as reference point. For BRZ, we use the size of the automaton after the first determinization phase as reference point. Furthermore, we abort a run if the runtime exceeds a previously set limit.

Visualization We visualize the runtime and overhead via cactus plots. Here, the x-axis shows the number of canonized systems ordered by the analyzed property, i.e., the further right, the worse the process performed. The sweet spot is the lower right, meaning that the worst runs still performed well. Note that cactus plots typically do not allow one to compare individual systems, as the fastest system for algorithm A may not be the fastest system for algorithm B.

5.2 Use Case 1: Walnut

For this use case, we look at a total of 52 systems that arose as intermediate calculations in Büchi-based arithmetic problems using the tool Walnut from [4, 7, 13, 15, 17, 26, 27, 29, 33, 35, 36, 37, 38, 39, 40, 41, 42] and personal communications with Jeffrey Shallit. These were particularly intensive calculations in Walnut, some running for days [41] and some using terabytes of memory [13]. Some could not be solved directly in Walnut ([7, 13, 39] for example). In particular, the benchmark includes systems such as `tribpseudo3` (“Open Problem 30” in [26]) which has not been successfully canonized yet. The minimum, median, maximum, and (rounded) mean number of states of the considered systems are 64, 9824, 60317, and 10113 whereas the minimum, median, maximum, and (rounded) mean number of input symbols are 2, 8, 1323, and 89. For this benchmark, we set a timeout of one hour.

Evaluation [Figure 1](#) and [Table 2](#) show the results of the benchmark. For the non-simulation-including configurations, we see that OTF, compared to SC, reduces the overhead of the intermediate automaton models. While the additional work of intermediate minimizations results in somewhat similar runtimes for the best cases, its positive impact on the more complex systems is evident as OTF canonizes notably more systems within the set timeout than SC.

For the BRZ-based approaches, we see that the general performance on the faster systems is better, which can be explained by the reduced overhead of

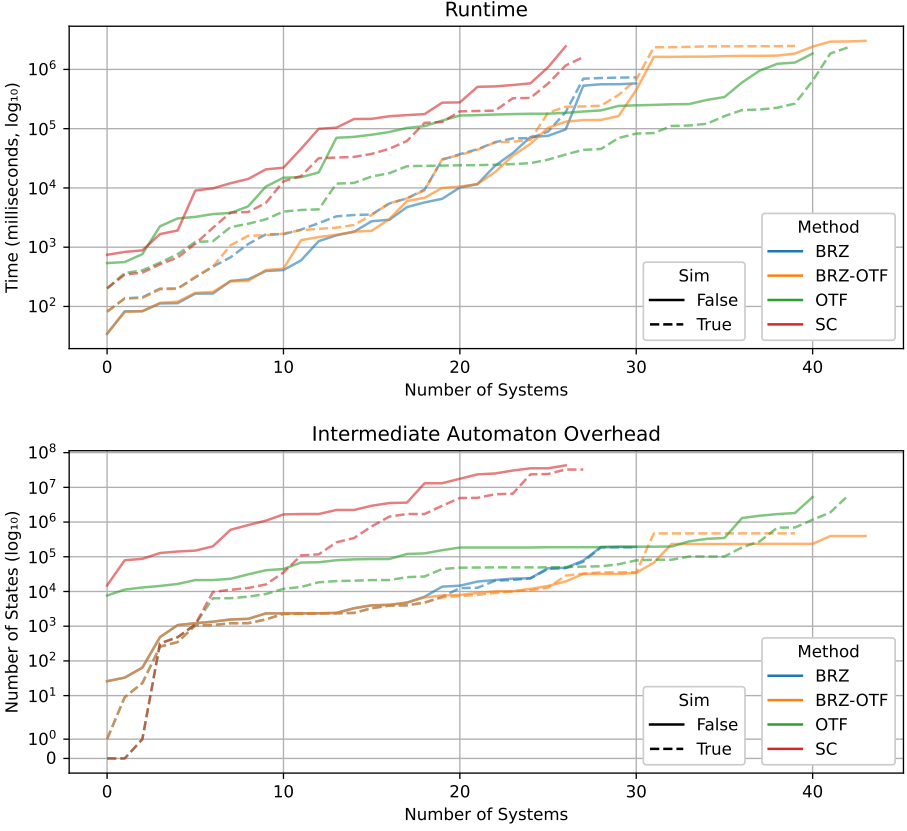


Fig. 1. Survival plots for the runtime and intermediate automaton overhead of the Walnut examples.

intermediate models. Still, more complex systems time out in the first phase. By compressing the intermediate models, BRZ-OTF significantly canonizes more systems compared to plain BRZ.

When looking at the simulation-including configurations, we see two diverging developments. For SC and OTF, simulation boosts performance both by improving runtime and reducing overhead, sometimes to the point of not introducing any redundant states at all. For the two BRZ-based configurations, simulation introduces a small increase in runtime and impacts the overhead only very little. An interesting comparison can be made between OTF-S and the BRZ-OTF approaches. While OTF-S sees an increased overhead for most systems, it yields better worst-case runtime performance than BRZ-OTF for cases that do not time out. Here, the single-phase OTF-S beats the double-phase BRZ-OTF(-S) despite greater resource consumption. Furthermore, it is worth noting that although OTF-S and BRZ-OTF successfully canonize a similar number of

systems (44 versus 45, respectively), these are not the same systems. Only two systems cannot be successfully canonized by either (among which is the previously mentioned `tribpseudo3`).

Table 2 gives additional insights into the behavior of the adaptive threshold. The gap between plain OTF and BRZ-OTF supports the previous observations that the intermediate models of the BRZ-based approaches are comparatively small and it is likely the second phase that causes timeouts. For both configurations, including simulation information reduces the number of minimizations, which highlights how equivalence registries enable one to incorporate external information that can further reduce the exploration space of the DFAs.

Overall, the results demonstrate the positive impact that OTF can have on the canonization process as well as its flexibility by being able to boost the performance of two different algorithmic approaches.

5.3 Use Case 2: Random Systems with Modular Structure

For this use case, we look at synthetic systems in order to analyze properties of OTF at different scales. Random models like the ones by Tabakov-Vardi [46] typically lack any structure which is what OTF is designed to exploit. Thus, we adapt the classic Tabakov-Vardi random NFA model by introducing structural constraints based on a modular partition of the state space. This is similar in spirit to Tabakov’s own random models with linear or grid structure [45].

Given an automaton with states $\{0, \dots, n-1\}$, we define an alphabet of size $k = \max(1, \lfloor \sqrt{n} \rfloor)$ and partition the states into k disjoint classes $\{C_0, \dots, C_{k-1}\}$ such that $q \in C_i \iff (q \bmod k) = i$. For each class C_i , we designate one initial and accepting state. For every symbol $a \in \{0, \dots, k-1\}$, transitions on a from states in class C_i are directed exclusively to states in class $C_{(i+a) \bmod k}$, introducing a modular structure that couples symbols and classes. Outgoing transitions are generated independently per source state with an expected out-degree given by the *density* parameter ρ and further restricted so that each class and symbol combination maintains a roughly balanced in- and out-degree distribution.

This model preserves the expected edge density of the Tabakov-Vardi random construction while embedding a modular structure. For the benchmark, we fix the transition density at 2 as we have observed that other densities result in similar behavior. We look at NFAs with $n \in \{20, 30, \dots, 300\}$ states and for each n , generate 10 automata with different random seeds each. For this benchmark, we set a timeout of 1000 seconds.

Evaluation Figure 2 and Table 3 show the results of the benchmark. Note that for the generated models, quotienting via simulation equivalence typically has no effect as states in this model will never exhibit similarity, except for some corner cases introduced by our preprocessing steps. As a result, simulation mostly manifests as noise in the runtime and intermediate automaton overhead, most notably on single-state automata where the overhead is zero.

The step-like pattern corresponds to the increases in system size where a growing number of states correlates with a longer duration for NFA canonization.

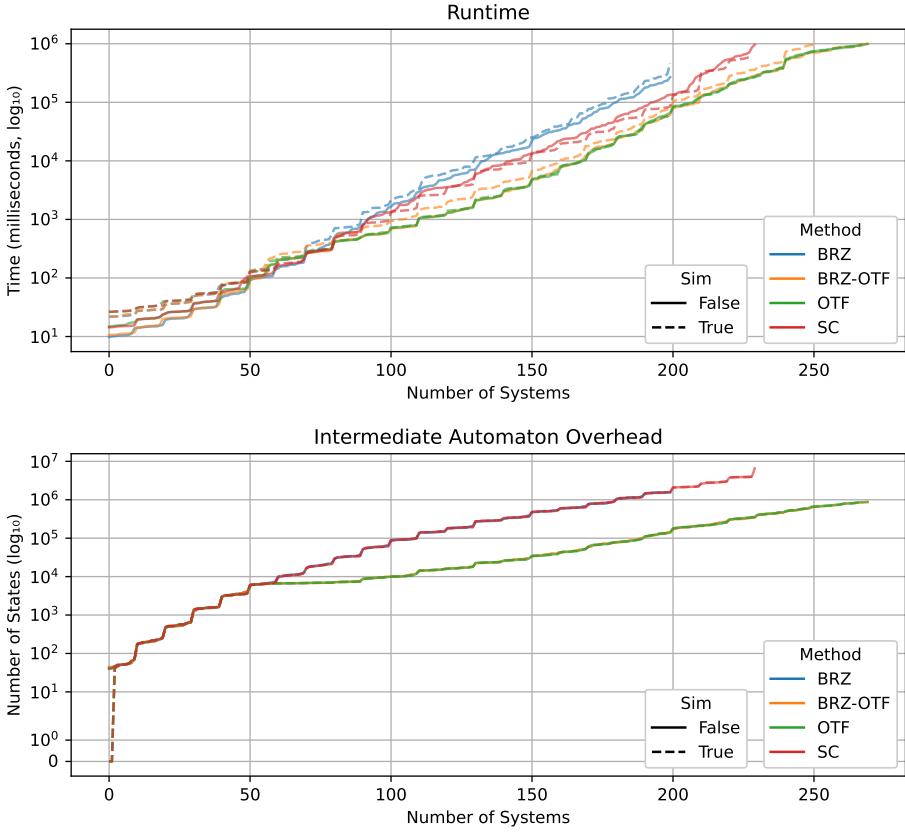


Fig. 2. Survival plots for the runtime and intermediate automaton overhead of the random systems with modular structure.

Table 3. Number of times that the adaptive threshold triggered a minimization for the random systems with modular structure.

	min	median	max	average
OTF	0	6	129	25.48
OTF-S	0	6	129	24.78
BRZ-OTF	0	6	135	25.50
BRZ-OTF-S	0	5	119	18.60

Similar to [Section 5.2](#), we can see that the BRZ-based approaches are competitive at their fastest performances but begin to struggle in the more complex scenarios. Only BRZ-OTF breaks this pattern and aligns with the performance of regular OTF. An explanation for this is given by the automaton overhead plot which shows that after a certain size, the OTF-flavored techniques are able to exploit the structure of the modular systems and the intermediate minimizations are

able to reduce the overall overhead. This also yields improved runtimes, though less dramatically, because the equivalence classes in this model are dominated by one class with a complex antichain, leading to slower searches.

Table 3 gives further insights into the behavior of the threshold. Compared to Section 5.2, the differences between the configurations are rather miniscule, aligning with the previous observations that OTF itself is the key performance booster. Again, OTF reduces overhead and improves runtime while simultaneously being flexible enough to integrate in different algorithmic concepts.

6 Conclusion and Future Work

In this paper, we present OTF – a framework for *on-the-fly* NFA canonization. Central to OTF are *equivalence registries* which connect the determinization phase with the minimization phase to enable performance improvements. With CCL and CCLS, we have presented two exemplary implementations of equivalence registries. Our performance evaluation shows that for NFAs with “structure”, the discussed OTF configurations outperform classic subset construction and Brzozowski’s algorithm for complex systems while being competitive for the remaining ones. We implement our approach in an open-source library [30] that is based on AutomataLib [3] and included in the theorem prover Walnut [50] since version 7.

Looking forward, there exist several directions for extension. One obvious direction deals with exploring further implementations of equivalence registries. Bonchi and Pous establish the notion of *up-to congruences* [6] which capture additional relations between equivalence classes, thus further reducing exploration space. However, in our initial experimentation, we observed a worse performance which aligns with results of others in this direction [16]. This appeared to be due to high-cardinality equivalence classes which are both slow and unlikely to match a given metastate. A potential direction for future research is the development of hybrid approaches that apply *up-to congruences* only to smaller cardinalities, or alternatively, exploit hierarchical structures. These ideas touch on the vast field of formal concept analysis [5].

Furthermore, the presented registries of this paper may be improved. States and transitions may be represented as binary decision diagrams [11, 47], potentially combined with a semi-symbolic approach [51]. This may be helpful in cases where systems have large alphabets [16]. The ability to encapsulate these technical details in equivalence registries makes OTF a flexible and powerful framework for future research.

Acknowledgments We thank Richard Mayr for helpful advice regarding simulations and Jeffrey Shallit for discussions regarding Walnut.

Disclosure of Interests Markus Frohme is funded by Deutsche Forschungsgemeinschaft (DFG), Grant 528775176. The authors are directly involved with development of the presented tool, and declare they have no financial interests.

Data-Availability Statement

The systems used in the experiments, the full results, and an executable Docker image for reproducing the data are available in Zenodo with identifier <https://doi.org/10.5281/zenodo.18163403>. For future use, the tool is maintained at <https://github.com/jn1z/OTF>.

References

- [1] Parosh Aziz Abdulla et al. “When Simulation Meets Antichains”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by Javier Esparza and Rupak Majumdar. Vol. 6015. Lecture Notes in Computer Science. Springer, 2010, pp. 158–174. DOI: [10.1007/978-3-642-12002-2_14](https://doi.org/10.1007/978-3-642-12002-2_14).
- [2] Alfred V. Aho and Margaret J. Corasick. “Efficient String Matching: An Aid to Bibliographic Search”. In: *Commun. ACM* 18.6 (1975), pp. 333–340. DOI: [10.1145/360825.360855](https://doi.org/10.1145/360825.360855).
- [3] *AutomataLib GitHub page*. Online. Accessed: Jan 12, 2026. URL: <https://github.com/LearnLib/automatalib>.
- [4] Jason P. Bell, Thomas F. Lidbetter, and Jeffrey O. Shallit. “Additive Number Theory via Approximation by Regular Languages”. In: *Int. J. Found. Comput. Sci.* 31.6 (2020), pp. 667–687. DOI: [10.1142/S0129054120410014](https://doi.org/10.1142/S0129054120410014).
- [5] Karell Bertet et al. “Lattices, closures systems and implication bases: A survey of structural aspects and algorithms”. In: *Theor. Comput. Sci.* 743 (2018), pp. 93–109. DOI: [10.1016/J.TCS.2016.11.021](https://doi.org/10.1016/J.TCS.2016.11.021).
- [6] Filippo Bonchi and Damien Pous. “Checking NFA equivalence with bisimulations up to congruence”. In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 457–468. DOI: [10.1145/2429069.2429124](https://doi.org/10.1145/2429069.2429124).
- [7] Wieb Bosma et al. “Using Walnut to solve problems from the OEIS”. In: *J. Integer Seq.* 28.1 (2025), p. 25.3.8. URL: <https://cs.uwaterloo.ca/journals/JIS/VOL28/Fokkink/fokkink9.html>.
- [8] Manfred Broy et al. *Model-Based Testing of Reactive Systems*. Vol. 3472. Lecture Notes in Computer Science. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. DOI: [10.1007/b137241](https://doi.org/10.1007/b137241).
- [9] Janusz A. Brzozowski. “Canonical regular expressions and minimal state graphs for definite events”. In: *Proc. Symposium of Mathematical Theory of Automata*. 1962, pp. 529–561.

- [10] Michaël Cadilhac et al. “Data Structures for Finite Downsets of Natural Vectors: Theory and Practice”. In: *Automated Technology for Verification and Analysis - 23rd International Symposium, ATVA 2025, Bengaluru, India, October 27-31, 2025, Proceedings*. Ed. by Meenakshi D’Souza, Raghavan Komondoor, and B. Srivathsan. Vol. 16145. Lecture Notes in Computer Science. Springer, 2025, pp. 425–446. DOI: [10.1007/978-3-032-08707-2_20](https://doi.org/10.1007/978-3-032-08707-2_20).
- [11] David Chocholatý et al. “Mata: A Fast and Simple Finite Automata Library”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II*. Ed. by Bernd Finkbeiner and Laura Kovács. Vol. 14571. Lecture Notes in Computer Science. Springer, 2024, pp. 130–151. DOI: [10.1007/978-3-031-57249-4_7](https://doi.org/10.1007/978-3-031-57249-4_7).
- [12] Lorenzo Clemente and Richard Mayr. “Efficient reduction of nondeterministic automata with application to language inclusion testing”. In: *Log. Methods Comput. Sci.* 15.1 (2019). DOI: [10.23638/LMCS-15\(1:12\)2019](https://doi.org/10.23638/LMCS-15(1:12)2019).
- [13] James D. Currie et al. “Properties of a ternary infinite word”. In: *RAIRO Theor. Informatics Appl.* 57 (2023), p. 1. DOI: [10.1051/ITA/2022010](https://doi.org/10.1051/ITA/2022010).
- [14] Laurent Doyen and Jean-François Raskin. “Antichain Algorithms for Finite Automata”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by Javier Esparza and Rupak Majumdar. Vol. 6015. Lecture Notes in Computer Science. Springer, 2010, pp. 2–22. DOI: [10.1007/978-3-642-12002-2_2](https://doi.org/10.1007/978-3-642-12002-2_2).
- [15] Gabriele Fici and Jeffrey O. Shallit. “Properties of a class of Toeplitz words”. In: *Theor. Comput. Sci.* 922 (2022), pp. 1–12. DOI: [10.1016/J.TCS.2022.04.006](https://doi.org/10.1016/J.TCS.2022.04.006).
- [16] Chen Fu et al. “On Equivalence Checking of Nondeterministic Finite Automata”. In: *Dependable Software Engineering. Theories, Tools, and Applications - Third International Symposium, SETTA 2017, Changsha, China, October 23-25, 2017, Proceedings*. Ed. by Kim Guldstrand Larsen, Oleg Sokolsky, and Ji Wang. Vol. 10606. Lecture Notes in Computer Science. Springer, 2017, pp. 216–231. DOI: [10.1007/978-3-319-69483-2_13](https://doi.org/10.1007/978-3-319-69483-2_13).
- [17] Daniel Gabric and Jeffrey O. Shallit. “The simplest binary word with only three squares”. In: *RAIRO Theor. Informatics Appl.* 55 (2021), pp. 1–7. DOI: [10.1051/ITA/2021001](https://doi.org/10.1051/ITA/2021001).
- [18] Hubert Garavel and Frédéric Lang. “Equivalence Checking 40 Years After: A Review of Bisimulation Tools”. In: *Lecture Notes in Computer Science* 13560 (2022). Ed. by Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos, pp. 213–265. DOI: [10.1007/978-3-031-15629-8_13](https://doi.org/10.1007/978-3-031-15629-8_13).
- [19] Rob J. van Glabbeek. “The Linear Time-Branching Time Spectrum (Extended Abstract)”. In: *CONCUR ’90, Theories of Concurrency: Unifica-*

- tion and Extension*, Amsterdam, The Netherlands, August 27-30, 1990, *Proceedings*. Ed. by Jos C. M. Baeten and Jan Willem Klop. Vol. 458. Lecture Notes in Computer Science. Springer, 1990, pp. 278–297. DOI: [10.1007/BFB0039066](https://doi.org/10.1007/BFB0039066).
- [20] Rob J. van Glabbeek and Bas Ploeger. “Five Determinisation Algorithms”. In: *Implementation and Applications of Automata, 13th International Conference, CIAA 2008, San Francisco, California, USA, July 21-24, 2008. Proceedings*. Ed. by Oscar H. Ibarra and Bala Ravikumar. Vol. 5148. Lecture Notes in Computer Science. Springer, 2008, pp. 161–170. DOI: [10.1007/978-3-540-70844-5_17](https://doi.org/10.1007/978-3-540-70844-5_17).
 - [21] Markus Holzer and Martin Kutrib. “Descriptive and computational complexity of finite automata - A survey”. In: *Inf. Comput.* 209.3 (2011), pp. 456–470. DOI: [10.1016/J.IC.2010.11.013](https://doi.org/10.1016/J.IC.2010.11.013).
 - [22] John Hopcroft. “An $n \log n$ algorithm for minimizing states in a finite automaton”. In: *Theory of machines and computations*. Elsevier, 1971, pp. 189–196.
 - [23] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. 2007.
 - [24] David Lee and Mihalis Yannakakis. “Principles and Methods of Testing Finite State Machines - a Survey”. In: *Proc. IEEE* 84.8 (1996), pp. 1090–1123. DOI: [10.1109/5.533956](https://doi.org/10.1109/5.533956).
 - [25] Yongming Luo. “Designing algorithms for big graph datasets : a study of computing bisimulation and joins”. PhD thesis. Technische Universiteit Eindhoven, Netherlands, 2015. ISBN: 978-90-386-3801-0.
 - [26] Joseph Meleshko et al. “Pseudoperiodic Words and a Question of Shevelev”. In: *Discret. Math. Theor. Comput. Sci.* 25.2 (2023). DOI: [10.46298/DMTCS.9919](https://doi.org/10.46298/DMTCS.9919).
 - [27] Bastien Mignoty et al. “Automatic proofs in combinatorial game theory”. 2024. URL: <https://hdl.handle.net/2268/323845>.
 - [28] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989. ISBN: 978-0-13-115007-2.
 - [29] Hamoon Mousavi and Jeffrey O. Shallit. “Mechanical Proofs of Properties of the Tribonacci Word”. In: *Combinatorics on Words - 10th International Conference, WORDS 2015, Kiel, Germany, September 14-17, 2015, Proceedings*. Ed. by Florin Manea and Dirk Nowotka. Vol. 9304. Lecture Notes in Computer Science. Springer, 2015, pp. 170–190. DOI: [10.1007/978-3-319-23660-5_15](https://doi.org/10.1007/978-3-319-23660-5_15).
 - [30] *OTF GitHub page*. Online. Accessed: Jan 12, 2026. URL: <https://github.com/jnlz/OTF>.
 - [31] Michael O. Rabin and Dana S. Scott. “Finite Automata and Their Decision Problems”. In: *IBM J. Res. Dev.* 3.2 (1959), pp. 114–125. DOI: [10.1147/RD.32.0114](https://doi.org/10.1147/RD.32.0114).
 - [32] Francesco Ranzato and Francesco Tapparo. “A New Efficient Simulation Equivalence Algorithm”. In: *22nd IEEE Symposium on Logic in Computer*

- Science (LICS 2007)*, 10-12 July 2007, Wroclaw, Poland, *Proceedings*. IEEE Computer Society, 2007, pp. 171–180. DOI: [10.1109/LICS.2007.8](https://doi.org/10.1109/LICS.2007.8).
- [33] Luke Schaeffer and Jeffrey Shallit. “The first-order theory of binary overlap-free words is decidable”. In: *Canadian Journal of Mathematics* 76.4 (2024), pp. 1144–1162. DOI: [10.4153/S0008414X23000342](https://doi.org/10.4153/S0008414X23000342).
- [34] Jeffrey Shallit. *Adventures with an Automatic Prover*. Online. Accessed: April 23, 2025. 2025. URL: <https://cs.uwaterloo.ca/~shallit/Talks/simons3.pdf>.
- [35] Jeffrey Shallit. *Developing Walnut Commands for Sequence Properties*. Online. Accessed: March 23, 2025. 2024. URL: <https://cs.uwaterloo.ca/~shallit/Talks/walnut-properties.pdf>.
- [36] Jeffrey Shallit. *The Logical Approach to Automatic Sequences*. Vol. 482. Cambridge University Press, 2022.
- [37] Jeffrey Shallit. *The Walnut Tutorial: Using A Tool for Doing Combinatorics on Words*. Online. Accessed: March 23, 2025. 2021. URL: <https://cs.uwaterloo.ca/~shallit/Talks/walnut-unpaused.pdf>.
- [38] Jeffrey Shallit. *Walnut Papers and Books (plus addenda)*. Online. Accessed: April 23, 2025. 2025. URL: <https://cs.uwaterloo.ca/~shallit/walnut-papers.html>.
- [39] Jeffrey O. Shallit. “The Narayana Morphism and Related Words”. In: *CoRR* abs/2503.01026 (2025). DOI: [10.48550/ARXIV.2503.01026](https://doi.org/10.48550/ARXIV.2503.01026). arXiv: [2503.01026](https://arxiv.org/abs/2503.01026).
- [40] Jeffrey O. Shallit, Sonja Linghui Shan, and Kai-Hsiang Yang. “Automatic sequences in negative bases and proofs of some conjectures of Shevelev”. In: *RAIRO Theor. Informatics Appl.* 57 (2023), p. 4. DOI: [10.1051/ITA/2022011](https://doi.org/10.1051/ITA/2022011).
- [41] Jeffrey O. Shallit, Arseny M. Shur, and Stefan Zorcic. “Power-free complementary binary morphisms”. In: *J. Comb. Theory, Ser. A* 207 (2024), p. 105910. DOI: [10.1016/J.JCTA.2024.105910](https://doi.org/10.1016/J.JCTA.2024.105910).
- [42] Jeffrey O. Shallit and Ramin Zarifi. “Circular critical exponents for Thue-Morse factors”. In: *RAIRO Theor. Informatics Appl.* 53.1-2 (2019), pp. 37–49. DOI: [10.1051/ITA/2018008](https://doi.org/10.1051/ITA/2018008).
- [43] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997. ISBN: 978-0-534-94728-6.
- [44] Vlad Slavici et al. “Finding the Minimal DFA of Very Large Finite State Automata with an Application to Token Passing Networks”. In: *CoRR* abs/1103.5736 (2011). arXiv: [1103.5736](https://arxiv.org/abs/1103.5736).
- [45] Deian Tabakov. “Experimental evaluation of explicit and symbolic automata-theoretic algorithms”. Master’s thesis. Rice University, USA, 2006. URL: <https://hdl.handle.net/1911/17923>.
- [46] Deian Tabakov and Moshe Y. Vardi. “Experimental Evaluation of Classical Automata Constructions”. In: *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*. Ed. by Geoff Sut-

- cliffe and Andrei Voronkov. Vol. 3835. Lecture Notes in Computer Science. Springer, 2005, pp. 396–411. DOI: [10.1007/11591191_28](https://doi.org/10.1007/11591191_28).
- [47] Valentin Touzeau et al. “Fast and exact analysis for LRU caches”. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 54:1–54:29. DOI: [10.1145/3290367](https://doi.org/10.1145/3290367).
 - [48] Mark Utting, Alexander Pretschner, and Bruno Legeard. “A taxonomy of model-based testing approaches”. In: *Softw. Test. Verification Reliab.* 22.5 (2012), pp. 297–312. DOI: [10.1002/STVR.456](https://doi.org/10.1002/STVR.456).
 - [49] Antti Valmari. “Simple Bisimilarity Minimization in $O(m \log n)$ Time”. In: *Fundam. Informaticae* 105.3 (2010), pp. 319–339. DOI: [10.3233/FI-2010-369](https://doi.org/10.3233/FI-2010-369).
 - [50] *Walnut GitHub page*. Online. Accessed: Jan 12, 2026. URL: <https://github.com/Walnut-Theorem-Prover>.
 - [51] Martin De Wulf et al. “Antichains: A New Algorithm for Checking Universality of Finite Automata”. In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 17–30. DOI: [10.1007/11817963_5](https://doi.org/10.1007/11817963_5).

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Quantum



Equivalence Checking of Quantum Circuits via Path-Sum and Weighted Model Counting

Wei-Jia Huang^{1*}, Christophe Chareton^{2**}, Yu-Fang Chen^{3***}, Kai-Min Chung³, Min-Hsiu Hsieh⁴, Alfons Laarman⁵, and Jingyi Mei^{†5}

¹ Quanta Computer Inc., Taoyuan, Taiwan

² CEA List, Paris-Saclay, France

³ Institute of Information Science, Academia Sinica, Taipei, Taiwan

⁴ Foxconn Research, Taipei, Taiwan

⁵ LIACS, Leiden University, Leiden, The Netherlands

Abstract. Equivalence checking of quantum circuits is a central verification task in quantum computing, ensuring the correctness of circuit optimizations, hardware mappings, and compilation pipelines. Among the primary symbolic methods for this purpose, the *path-sum formalism* provides a compact representation with powerful reduction rules that yield a canonical form for the classically simulable Clifford fragment, but *confluence* fails beyond the Clifford fragment. We introduce a new *weighted model counting* (WMC) encoding for path-sums and combine it with the existing path-sum reductions to obtain a verifier that is both complete and efficient. Our method applies reductions whenever possible and invokes the WMC-based decision procedure on the residual path-sum, yielding a complete semantic check up to a global phase. We implement the approach and evaluate it on standard benchmarks. Results show that the hybrid method outperforms either component in isolation and competes with state-of-the-art tools.

1 Introduction

Quantum computation stores and processes information in systems governed by quantum mechanics [34]. Quantum devices are rapidly maturing [6, 2]. Unlike classical data, quantum data cannot be cloned, and evolution is described by unitary operators. In practice, computations are specified by sequences of such unitaries, formalized as *quantum circuits*. Quantum circuits are typically executed on a coprocessor architecture: a classical host emits instructions that initialize a quantum register and apply these gates.

As these circuits are compiled for execution, they undergo successive transformations—optimizations, decompositions, and hardware mappings. Each transformation should preserve semantics, which makes *circuit equivalence* a central

* wei.jia.huang.physics@gmail.com. The work was conducted while he was a researcher at Foxconn Research.

** christophe.chareton@cea.fr

*** yfc@iis.sinica.edu.tw

† j.mei@liacs.leidenuniv.nl

verification task: checking whether two circuits generated at different stages of the toolchain induce the same unitary, possibly up to a global phase.

The exact circuit equivalence problem is **NQP**-hard [40], which makes a polynomial-time exhaustive algorithm unlikely. Work in this area has focused on two main approaches. The first is *equational reasoning*, which chooses a symbolic representation together with an equational theory and searches for a rewriting path between circuit representations [18, 17, 3, 41, 16]. This approach can be efficient when short proofs exist, but its completeness and confluence properties degrade outside restricted fragments. The second is *model-based reasoning*, which encodes equivalence as a decision problem on a semantic model and solves it with a complete, though often expensive, procedure [8, 38, 31, 13, 12, 1].

In this work, we study both approaches in concrete form. For equational reasoning, we use the *path-sum calculus* [3], an equational system that represents circuits as algebraic sums over computational paths. This enables algebraic rewriting rules to simplify path-sum formulae and prove equivalence when a short derivation exists. For model-based reasoning, we introduce a new use of *weighted model counting* (WMC). To highlight the contrast, we present two simple examples. The path-sum method shows how algebraic rewrites can prove that $H(1)H(1)$ equals the identity, where $H(1)$ means applying the Hadamard gate H to the first qubit. We then illustrate our WMC encoding using the same example for simplicity, noting that the same encoding also applies in cases where algebraic rewriting alone is insufficient.

Example 1 (Path-sum reasoning). Consider the single-qubit circuit $C = H(1)H(1)$, where H is the Hadamard gate. The path-sum semantics of $H(1)$ is

$$H(1): |x\rangle \mapsto \frac{1}{\sqrt{2}} \sum_{y \in \{0,1\}} (-1)^{x \cdot y} |y\rangle.$$

This matches the standard semantics of $H(1)$, which maps $|0\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. The path-sum formula describes how the basis state $|x\rangle$ contributes to each basis state $|0\rangle$ and $|1\rangle$ after the gate execution.

Composing two H gates (see formula 3 for formal definition) yields

$$HH: |x\rangle \mapsto \frac{1}{2} \sum_{y,z \in \{0,1\}} (-1)^{x \cdot y + y \cdot z} |z\rangle. \quad (1)$$

Using the $[HH]$ rewrite rules that will be introduced in Section 2.2, the above path-sum simplifies to $|x\rangle \mapsto |x\rangle$, proving that HH is the identity.

Before giving the corresponding WMC example, we recall the definition of weighted model counting.

Definition 1 (Weighted Model Counting). Let Γ be a Boolean formula over variables V , and let $W: V \times \{0,1\} \rightarrow \mathbb{A}$ be a weight function that assigns to each literal (v,b) a weight $W(v,b)$ in some commutative semiring $\mathbb{A}$. The weighted

model count of (Γ, W) is defined as

$$\text{WMC}(\Gamma, W) = \sum_{\tau \models \Gamma} \prod_{v \in V} W(v, \tau(v)),$$

where the sum ranges over all satisfying assignments $\tau : V \rightarrow \{0, 1\}$ of Γ .

Example 2 (Reducing Path-Sum Reasoning to Weighted Model Counting). Rather than applying rewrite rules, we encode the problem of checking whether the path-sum for $H(1)H(1)$ equals the identity as a weighted model counting instance.

We start with the original variables x, y, z and introduce two auxiliary Boolean variables s_1 and s_2 , representing the products $x \cdot y$ and $y \cdot z$, respectively. The Boolean formula fed into the weighted model counter is

$$\Gamma \equiv (s_1 \leftrightarrow (x \wedge y)) \wedge (s_2 \leftrightarrow (y \wedge z)) \wedge (x \leftrightarrow z).$$

The weight function is defined as follows. For Boolean variable $b \in \{0, 1\}$,

$$W(v, b) = \begin{cases} 1 & \text{if } v \in \{x, y, z\}, \\ (-1)^b & \text{if } v \in \{s_1, s_2\}. \end{cases}$$

Below, we explain the idea behind the encoding Γ and weight function W . First, observe that all integer variables in Equation (1) are over the domain $\{0, 1\}$, in this introduction, we slightly abuse the notation and treat them simultaneously as Boolean and integer variables. So the two Boolean constraints $s_1 \leftrightarrow (x \wedge y)$ and $s_2 \leftrightarrow (y \wedge z)$ also have a corresponding integer interpretation $s_1 = x \cdot y$ and $s_2 = y \cdot z$. Hence the amplitude of $|z\rangle$ in Equation (1) can be rewritten as

$$\frac{1}{2} \sum_{y, z \in \{0, 1\}} (-1)^{x \cdot y + y \cdot z} = \frac{1}{2} \sum_{y, z \in \{0, 1\}} (-1)^{s_1} \cdot (-1)^{s_2}.$$

From the right-hand side of the above equation, it suffices to consider only the contributions of s_1 and s_2 to the amplitude of the target basis $|z\rangle$. Accordingly, we assign the weight $(-1)^{b_i}$ to the corresponding literal (s_i, b_i) , while all other literals are assigned weight one.

Finally, the constraint $x \leftrightarrow z$ ensures that we only consider amplitudes where $|x\rangle$ contributes to itself in the path-sum, i.e., the diagonal entries of the corresponding unitary matrix. It follows that the path-sum equals the identity up to global phase if and only if the absolute value of the weighted count is $2^1 \times e^{i\theta}$, where $e^{i\theta}$ is the *global phase*, and this example has only one qubit, which matches the diagonal sum of a 2×2 identity matrix.

To summarize, we propose a hybrid method that integrates equational reasoning with model-based reasoning. Our method applies reductions whenever possible and invokes the WMC-based decision procedure on the residual path-sum, yielding a complete semantic check up to a global phase. Our approach is built on the *path-sum* framework [3], where the unitary evolution of a quantum system from state A to state B is expressed as a weighted sum over all possible paths. Our main contributions are:

- A novel weighted model counting (WMC) encoding of the path-sum equivalence problem in the sense of Amy [3], enabling a complete semantic check.
- A hybrid verification tool that combines path-sum reductions with the WMC-based procedure.
- An implementation and experimental evaluation on standard benchmarks, demonstrating that the hybrid approach consistently outperforms either reductions or WMC alone, and improves over simple strategy combinations.

Related work. Path-sum reduction produces phase polynomials with symbolic path conditions, a structured algebraic object that cannot be encoded using any existing WMC formulation, including Quokka#’s per-gate encoding for quantum circuit equivalence testing [30–32].

Our contribution is a new encoding that translates these reduced forms into weighted Boolean variables and constraints. This allows WMC to operate after reduction, which was previously impossible. Without this encoding, model counting cannot even express the intermediate objects produced by path-sum.

Two main lines of research exist for quantum circuit equivalence checking: *equational reasoning* and *model-based approaches*.

Equational reasoning frameworks, such as path-sum [3–5, 9, 37] and the ZX-calculus [18, 19, 27, 20], establish equivalence by algebraic or diagrammatic rewriting. These methods are sound, and in principle, they can be made complete. Yet completeness forces reductions to raw matrix operations, which are only of theoretical interest. Practical tools, therefore, trade completeness for efficiency, applying partial but efficient reasoning procedures.

Model-based approaches, by contrast, construct explicit symbolic models of quantum states and circuits using decision diagrams [8, 25, 24, 10], tree automata [12, 14, 1, 11], SMT [7, 15], or weighted model counting [30–32]. These techniques are typically complete. They may be slower than equational methods on simple circuits, but they scale more robustly on challenging cases, often outperforming equational methods when completeness becomes impractical.

Combining equational reasoning with model-based approaches offers the potential for better synergy. An early attempt was made by QCEC [8], which combines ZX-calculus with decision diagrams in a portfolio approach. This design, however, runs two solvers in parallel without interaction. Another approach to combine both paradigms appeared in the context of symbolic variational circuits [35]. This approach complements the incomplete ZX approach with a complete method, but this approach has not yet been evaluated on general circuit equivalence checking.

2 Background

2.1 Quantum preliminaries

We present only the essentials due to space constraints; for a comprehensive treatment, see [34].

A one-qubit quantum state is a unit vector $[\alpha_0^{\alpha_1}] \in \mathbb{C}^2$ written in Dirac notation as $\alpha_0|0\rangle + \alpha_1|1\rangle$ with $|\alpha_0|^2 + |\alpha_1|^2 = 1$. An n -qubit quantum state can be written as $|\psi\rangle = \sum_{x \in \{0,1\}^n} \alpha_x |x\rangle$, with the sum of absolute square value of all probabilistic amplitudes $\sum_{x \in \{0,1\}^n} |\alpha_x|^2 = 1$. The dynamics of a quantum state are governed by *unitary operators*. A unitary U is a linear map on $\mathcal{H}$ with $U^\dagger U = I$, where $U^\dagger$ denotes the *conjugate transpose* of U . Applying the unitary gate U to $|\psi\rangle$ produces the state $U|\psi\rangle$. Elementary quantum *gates* are fixed unitaries acting on a small number of qubits (one or two). Common examples include the Hadamard gate $H(1)|x\rangle = \frac{1}{\sqrt{2}}(|0\rangle + (-1)^x|1\rangle)$, the controlled-NOT gate $CNOT(1,2)|x,z\rangle = |x, x \oplus z\rangle$, the rotation gate $Rz(\theta)(1)|x\rangle = e^{i\theta \cdot x}|x\rangle$, and the controlled-rotation gate $CRz(\theta)(1,2)|x,z\rangle = e^{i\theta \cdot x \cdot z}|x,z\rangle$.

A *quantum circuit* is a finite sequence of gates; each gate is a unitary acting on one or more qubits, and the whole circuit denotes a unitary matrix $U \in \mathbb{C}^{2^n \times 2^n}$. Parallel composition of two circuits corresponds to their *tensor product* and sequential composition to their *matrix multiplication*. Two circuits C_1, C_2 on the same n qubits are *equivalent* if $U[C_1] = U[C_2]$, where $U[C]$ means the unitary matrix of the circuit C . They are *equivalent up to global phase* if $U[C_1] = e^{i\theta}U[C_2]$ for some real θ . This is the standard notion for circuit equivalence checking. Equivalently, to check if two circuits C_1 and C_2 are equivalent, it suffices to verify that $U[C_2]U[C_1]^\dagger = I$. They are equivalent up to global phase iff $U[C_2]U[C_1]^\dagger = e^{i\theta}I$ for some real θ .

2.2 Path-sum formalism

Path-sums provide a symbolic representation of a quantum circuit's semantics as sums over Boolean "paths" [3, 4]. Formally, for a sequence of state variables $\mathbf{x} = \{x_1, \dots, x_n\}$ (where n is the number of qubits), a path-sum is a triple $\mathcal{P} = \langle \mathbf{y}, \Phi, \mathcal{O} \rangle$ where (i) $\mathbf{y} = \{y_1, \dots, y_m\}$ is the set of Boolean *path variables*, (ii) $\Phi : \mathbb{B}^{|\mathbf{x}|+|\mathbf{y}|} \mapsto \mathbb{R}$ is a phase polynomial with real coefficients, and (iii) $\mathcal{O} : \mathbb{B}^{|\mathbf{x}|+|\mathbf{y}|} \mapsto \mathbb{B}^{|\mathbf{x}|}$ is an output function that maps path and state variables to a Boolean vector describing the output state. Its action on a basis state $|\mathbf{x}\rangle$ is given as

$$|\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2^{|\mathbf{y}|}}} \sum_{\mathbf{y} \in \{0,1\}^{|\mathbf{y}|}} e^{i\Phi(\mathbf{x},\mathbf{y})} |\mathcal{O}(\mathbf{x},\mathbf{y})\rangle. \quad (2)$$

Elementary gates admit concise path-sum rules. For example, writing z_k to denote the k th element of a vector $\mathbf{z}$, we have:

$$\begin{aligned} H(t) : \langle \mathbf{y}, \Phi, \mathcal{O} \rangle &\mapsto \langle \mathbf{y} \cup \{y_f\}, \Phi + \tilde{x}_t \tilde{y}_f \pi, \mathcal{O}[x_t \leftarrow y_f] \rangle, \quad (y_f \text{ is fresh}) \\ CNOT(c,t) : \langle \mathbf{y}, \Phi, \mathcal{O} \rangle &\mapsto \langle \mathbf{y}, \Phi, \mathcal{O}[x_t \leftarrow x_c \oplus x_t] \rangle, \\ Rz(\theta)(t) : \langle \mathbf{y}, \Phi, \mathcal{O} \rangle &\mapsto \langle \mathbf{y}, \Phi + \tilde{x}_t \theta, \mathcal{O} \rangle, \\ CRz(\theta)(c,t) : \langle \mathbf{y}, \Phi, \mathcal{O} \rangle &\mapsto \langle \mathbf{y}, \Phi + \tilde{x}_c \tilde{x}_t \theta, \mathcal{O} \rangle. \end{aligned}$$

where t denotes the target qubit, c denote the control qubit, $\tilde{b}$ denotes *bool-to-int casting*, interpreting XOR $\oplus$ as $\widetilde{b_i \oplus b_j} := \tilde{b}_i + \tilde{b}_j - 2 \cdot \tilde{b}_i \cdot \tilde{b}_j$ and NOT $\neg$ as

$\widetilde{\neg}b := 1 - \widetilde{b}$ ⁶ and

$$\mathcal{O}[x_t \leftarrow \sigma](\mathbf{x}, \mathbf{y})_j = \begin{cases} \mathcal{O}(\mathbf{x}, \mathbf{y})_j, & \text{if } t \neq j, \\ \sigma, & \text{if } t = j. \end{cases}$$

Example 3. The path-sum triple for the circuit $\mathbf{H}(1)\mathbf{H}(1)$ is constructed as follows:

$$\langle \emptyset, 0, |x\rangle \rangle \xrightarrow{\mathbf{H}(1)} \langle \{y_0\}, xy_0\pi, |y_0\rangle \rangle \xrightarrow{\mathbf{H}(1)} \langle \{y_0, y_1\}, xy_0\pi + y_0y_1\pi, |y_1\rangle \rangle.$$

Returning to Dirac notation, the action of the circuit on each basis state $|x\rangle$ is

$$|x\rangle \mapsto \frac{1}{\sqrt{2^{|\{y_0, y_1\}|}}} \sum_{y_0, y_1 \in \{0, 1\}} e^{i(xy_0\pi + y_0y_1\pi)} |y_1\rangle = \frac{1}{2} \sum_{y_0, y_1 \in \{0, 1\}} (-1)^{(xy_0 + y_0y_1)} |y_1\rangle,$$

which coincides with the expression in Example 1. $\square$

Composition of circuits corresponds to the symbolic composition of path-sums by substituting outputs and adding phases as follows.

$$\langle \mathbf{y}', \Phi', \mathcal{O}' \rangle \circ \langle \mathbf{y}, \Phi, \mathcal{O} \rangle = \langle \mathbf{y} \cup \mathbf{y}', \Phi(\mathbf{x}, \mathbf{y}) + \Phi'(\mathcal{O}(\mathbf{x}, \mathbf{y}), \mathbf{y}'), \mathcal{O}'(\mathcal{O}(\mathbf{x}, \mathbf{y}), \mathbf{y}') \rangle \quad (3)$$

The path-sum of the identity circuit is $\langle \emptyset, 0, |x\rangle \rangle$. Any quantum circuit can be expressed as a path-sum by starting from this identity form and composing it step by step with each gate in the circuit.

However, this gate-by-gate construction can yield different formulas representing the same vector semantics as in Equation (2). For instance, applying two consecutive Hadamard gates to the same qubit yields the path-sum

$$\langle \mathbf{y}, \Phi, \mathcal{O} \rangle \xrightarrow{\mathbf{H}(i)\mathbf{H}(i)} \langle \mathbf{y} \cup \{y_0, y_1\}, \Phi + x_i y_0 \pi + y_0 y_1 \pi, \mathcal{O}[x_i \leftarrow y_0][x_i \leftarrow y_1] \rangle$$

Here, y_0 and y_1 are fresh variables not appearing in the original path-sum.

Since the Hadamard operation is involutive, $\mathbf{HH}$ is equivalent to the identity, we also have the equivalent and much simpler path-sum

$$\langle \mathbf{y}, \Phi, \mathcal{O} \rangle \xrightarrow{\mathbf{H}(i)\mathbf{H}(i)} \langle \mathbf{y}, \Phi, \mathcal{O} \rangle$$

To formalize this equivalence, the path-sum framework is equipped with an *equational theory*, that is, a collection of reduction rules generating an equivalence relation compatible with the path-sum semantics.

The main reduction rules are shown in Fig. 1. Rule $[\mathbf{HH}]$ symbolically identifies and cancels pairs of paths that contribute equal amplitudes but with opposite phases. Similarly, rule $[\omega]$ captures a different kind of symbolic simplification: when two amplitude vectors differ by a $\pi/2$ phase, their sum corresponds to a

⁶ In practice, we omit the explicit casting notation for atomic binary integer values

$ \begin{array}{l} \text{[HH]} \frac{y_0 \notin (\text{var}(\mathcal{O}) \cup \text{var}(Q) \cup \text{var}(R)) \quad y_1 \notin \text{var}(Q)}{\langle \mathbf{y} \cup \{y_0, y_1\}, (\frac{1}{2}y_0(y_1 + Q) + R) \cdot 2\pi, \mathcal{O} \rangle \equiv \langle \mathbf{y}, R[y_1 \leftarrow \overline{Q}] \cdot 2\pi, \mathcal{O}[y_1 \leftarrow \overline{Q}] \rangle} \\ \\ \text{[}\omega\text{]} \frac{y_0 \notin (\text{var}(\mathcal{O}) \cup \text{var}(Q) \cup \text{var}(R))}{\langle \mathbf{y} \cup \{y_0\}, (\frac{1}{4}y_0 + \frac{1}{2}y_0Q + R) \cdot 2\pi, \mathcal{O} \rangle \equiv \langle (y), (\frac{1}{8} - \frac{1}{4}\overline{Q} + R) \cdot 2\pi, \mathcal{O} \rangle} \end{array} $

Fig. 1: The path-sum reduction rules [3], where Q is an integer formula and R a real formula. $\overline{Q}$ computes the modulo 2 integer value, and we also abuse the notation to denote the corresponding Boolean value, i.e., one maps to true and zero to false. We refer the readers to [3] for the $\overline{Q}$ construction.

bisector direction with normalized amplitude.⁷ The rule abstracts this geometric addition at the symbolic level. We refer readers to [3] for the detailed derivations of these equations. Readers may safely skip them on a first reading without losing track of our main technical contribution.

For *Clifford circuits*,⁸ the reduction rules from Fig. 1 are both *complete* (any two equivalent circuits can be reduced to the same normal form) and *confluent* (the reduction process always yields a unique normal form, regardless of the order of rule applications). Beyond that, for example, in the Z^* fragment,⁹ completeness can be obtained but confluence is lost [42]. This means that while equivalence is always provable in principle, there is no deterministic strategy to find a proof, and practical verification must rely on heuristics.

In this paper, we present a complete method for handling quantum circuits generated by any set of gates, including the Z^* fragment. We first apply the reduction rules (e.g., those in Fig. 1) to simplify the formula as far as possible. For the residual path-sum, we then show in the next section how equivalence checking (up to global phase) can be reduced to a weighted model counting problem.

3 Technical Contribution

Our goal is to verify the equivalence of two n -qubit circuits C_1, C_2 . Due to reversibility of the circuits, this reduces to checking whether

$$C := C_1 C_2^\dagger$$

⁷ If two complex amplitudes a and $ae^{i\pi/2}$ have equal magnitude but differ by a phase of 90° , their sum is $a(1+i) = \sqrt{2}ae^{i\pi/4}$. The resulting vector points in the direction halfway between them in the complex plane, called the *bisector*.

⁸ Clifford circuits are those generated by the Hadamard, Phase, and CNOT gates. They form the Clifford group, which is efficiently classically simulable [22].

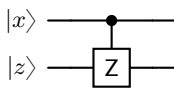
⁹ Circuits in gate set $\{H, CNOT, R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{pmatrix}\}$.

is equal to I (exact equivalence) or to $e^{i\theta}I$ for some real θ (equivalence up to global phase). We construct the path-sum $PS(C)$ and then proceed in two steps:

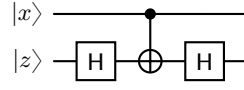
1. apply path-sum reductions (Fig. 1) to simplify $PS(C)$, and
2. encode the residual instance as a weighted model counting (WMC) problem and decide it exactly.

Step 1 largely follows the standard path-sum calculus, but requires some refinements. In particular, reduction rules depend on recognizing syntactic patterns in path-sum formulae, yet many equivalent expressions do not match these patterns directly. In practical use, one needs to add additional rules to reorganize the phase polynomials and Boolean output functions to enable pattern matching. We also slightly relaxed the constraints for $Rz(\theta)$ and $CRz(\theta)$ gate constructions to permit arbitrary rotation angles. These relaxations integrate smoothly with the path-sum formalism of [3], but, as expected, the reduction rules cannot handle interference on arbitrary angles and therefore remain incomplete. This motivates Step 2, where we derive a sound and complete equivalence-checking procedure for path-sum formulae through a reduction to WMC.

Example 4. We illustrate a simple case of gate simulation. The controlled rotation gate $CRz(\pi)$ (Fig. 2a) can be simulated by a CNOT gate conjugated with Hadamard gates on its target qubit (Fig. 2b). We refer to the circuit in Fig. 2a as C_1 and the one in Fig. 2b as C_2 . The two circuits C_1 and C_2 are *semantically equivalent*. To check the equivalence between C_1 and C_2 , since $U[C_2]^\dagger = U[C_2]$



(a) The controlled-rotation $C_1 := CRz(\pi)(1, 2)$



(b) The controlled Z rotation: Clifford simulation through circuit $C_2 := H(2)CNOT(1, 2)H(2)$

Fig. 2: Running example, simulating Control Z rotation with Clifford gates

based on its definition, Lemma 1 instantiates as:

$$C_1 \equiv C_2 \text{ iff for all } \mathbf{x} \in \{0, 1\}^2, PS(C_1 C_2^\dagger) |\mathbf{x}\rangle = |\mathbf{x}\rangle. \quad (4)$$

Based on the composition rule from Equation (3), we have the composed circuit in path-sum formalism as follows:

$$\begin{aligned}
 PS(C_1; C_2) &= \langle \emptyset, 0, |x, z\rangle \rangle \\
 &\xrightarrow{\text{CR}_Z(\pi)(1,2)} \langle \emptyset, \pi xz, |x, z\rangle \rangle \\
 &\xrightarrow{H(2)} \langle \{y_0\}, \pi(xz + zy_0), |x, y_0\rangle \rangle \\
 &\xrightarrow{\text{CNOT}(1,2)} \langle \{y_0\}, \pi(xz + zy_0), |x, x \oplus y_0\rangle \rangle \\
 &\xrightarrow{H(2)} \langle \{y_0, y_1\}, \pi(xz + zy_0 + (x \oplus y_0)y_1), |x, y_1\rangle \rangle \\
 &= \langle \{y_0, y_1\}, \pi(xz + zy_0 + (x + y_0 - 2xy_0)y_1), |x, y_1\rangle \rangle \quad (5)
 \end{aligned}$$

$$\equiv \langle \{y_0, y_1\}, \pi(xz + zy_0 + (x + y_0)y_1), |x, y_1\rangle \rangle \pmod{2\pi}. \quad (6)$$

Note that for the last step, since $e^{-2xy_0\pi} = 1$, we can further reduce the resulting path-sum from Equation (5) to Equation (6). To apply [HH] reduction, one has to rearrange the term $(xz + zy_0 + (x + y_0)y_1) \cdot \pi$ to be in the form of the polynomial $(\frac{1}{2}y_0(y_1 + z) + \frac{1}{2}x(y_1 + z)) \cdot 2\pi$, where $Q = z$, $R = \frac{1}{2}x(y_1 + z)$, $\mathcal{O}(\mathbf{x}, \mathbf{y}) = (x, y_1)$. So $y_0 \notin \text{var}(Q) \cup \text{var}(R) \cup \text{var}(\mathcal{O})$, the phase polynomial is then reduced to be $\langle \emptyset, R[y_1 \leftarrow \bar{Q}] \cdot 2\pi, \mathcal{O}[y_1 \leftarrow \bar{Q}] \rangle$, where $R[y_1 \leftarrow \bar{Q}] \cdot 2\pi = \frac{1}{2}x(z + z) \cdot 2\pi \equiv 0 \pmod{2\pi}$ and $\mathcal{O}(\mathbf{x}, \mathbf{y})[y_1 \leftarrow z] = |x, z\rangle$.

Thus we have the reduction: Equation (6) $\xrightarrow{[\text{HH}]}$ $\langle \emptyset, 0, |x, z\rangle \rangle$. $\square$

In [3], completeness is established only for the Clifford fragment of the calculus. To extend it to general quantum circuits, we reduce the problem to weighted model counting (WMC).

3.1 Encoding path-sum equivalence into WMC

We reduce equivalence to a check on diagonal amplitudes. For $C = C_2C_1^\dagger$ and basis $|\mathbf{x}\rangle$, we assume $PS(C)$ is

$$|\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2^{|\mathbf{y}|}}} \sum_{\mathbf{y} \in \{0,1\}^{|\mathbf{y}|}} e^{i\Phi(\mathbf{x}, \mathbf{y})} |\mathcal{O}(\mathbf{x}, \mathbf{y})\rangle.$$

and slightly abuse notation and write $PS(C)$ also for its *matrix representation*. Formally, it defines the matrix $\frac{1}{\sqrt{2^{|\mathbf{y}|}}} \sum_{\mathbf{y} \in \{0,1\}^{|\mathbf{y}|}} e^{i\Phi(\mathbf{x}, \mathbf{y})} |\mathcal{O}(\mathbf{x}, \mathbf{y})\rangle \langle \mathbf{x}|$.

Because $PS(C)$ is unitary, for each basis state $|\mathbf{x}\rangle$, the output $PS(C)|\mathbf{x}\rangle$ has L_2 -norm equals 1. Therefore $PS(C)|\mathbf{x}\rangle = |\mathbf{x}\rangle$ iff its *diagonal amplitude* satisfies $\langle \mathbf{x} | PS(C) | \mathbf{x} \rangle = 1$. For equivalence up to a global phase, it suffices to require $\langle \mathbf{x} | PS(C) | \mathbf{x} \rangle = e^{i\theta}$

Lemma 1. *$PS(C)$ is the identity if and only if, for every basis state $|\mathbf{x}\rangle$, its diagonal amplitude satisfies*

$$\langle \mathbf{x} | PS(C) | \mathbf{x} \rangle = 1.$$

Corollary 1. *Let $PS(C)$ act on n qubits. Then $PS(C)$ is equivalent to identity up to global phase if and only if*

$$\sum_{\mathbf{x} \in \{0,1\}^n} \langle \mathbf{x} | PS(C) | \mathbf{x} \rangle = 2^n \cdot e^{i\theta} \text{ for a given } \theta.$$

Proof. The “only if” direction is immediate: if $PS(C) = e^{i\theta} \cdot I$, for every $|\mathbf{x}\rangle$, $\langle \mathbf{x} | PS(C) | \mathbf{x} \rangle = e^{i\theta} \cdot \langle \mathbf{x} | \mathbf{x} \rangle = e^{i\theta}$, thus $\sum_{\mathbf{x} \in \{0,1\}^n} \langle \mathbf{x} | PS(C) | \mathbf{x} \rangle = 2^n \cdot e^{i\theta}$.

For the “if” direction, suppose $\sum_{\mathbf{x} \in \{0,1\}^n} \langle \mathbf{x} | PS(C) | \mathbf{x} \rangle = 2^n \cdot e^{i\theta}$. Since $PS(C)$ is unitary, for each basis state $\mathbf{x}$ we have $|\langle \mathbf{x} | PS(C) | \mathbf{x} \rangle| \leq 1$. Therefore,

$$2^n = \left| \sum_{\mathbf{x} \in \{0,1\}^n} \langle \mathbf{x} | PS(C) | \mathbf{x} \rangle \right| \leq \sum_{\mathbf{x} \in \{0,1\}^n} |\langle \mathbf{x} | PS(C) | \mathbf{x} \rangle| \leq \sum_{\mathbf{x} \in \{0,1\}^n} 1 = 2^n.$$

Since the first and last terms of the chain are equal, both inequalities must in fact be equalities, Thus $|\langle \mathbf{x} | PS(C) | \mathbf{x} \rangle| = 1$ for every $\mathbf{x}$ and since the modulus of total sum is the maximum 2^n , that global phase must be the same, i.e. $\langle \mathbf{x} | PS(C) | \mathbf{x} \rangle = c$ for all $\mathbf{x}$. So $\langle \mathbf{x} | e^{-i\theta} \cdot PS(C) | \mathbf{x} \rangle = 1$. Based on Lemma 1, we conclude $e^{-i\theta} \cdot PS(C) = I$ thus $PS(C) = e^{i\theta} \cdot I$. $\square$

Next we show how to use WMC to compute the sum of the diagonal values $\sum_{\mathbf{x} \in \{0,1\}^n} \langle \mathbf{x} | PS(C) | \mathbf{x} \rangle$. For the encoding, we first rewrite the *phase polynomial* $\Phi(\mathbf{x}, \mathbf{y})$ of $PS(C)$ to a finite sum of monomial terms

$$\Phi(\mathbf{x}, \mathbf{y}) = \sum_{j \in J} \text{Mon}_j, \quad \text{Mon}_j = k_j \cdot \text{bc}_j(\mathbf{x}, \mathbf{y}).$$

Here J is the set of monomial index, each Mon_j is the product of a real coefficient k_j and a Boolean condition $\text{bc}_j(\mathbf{x}, \mathbf{y})$. The Boolean condition indicates *when* this monomial contributes: it evaluates to 1 if the assignment $(\mathbf{x}, \mathbf{y})$ activates the term, and 0 otherwise.

Example 5. Considering the path-sum in Equation (5) of Example 4:

$$PS(C_2; C_1^\dagger) = \langle \{y_0, y_1\}, (xz + zy_0 + (x + y_0 - 2xy_0)y_1) \cdot \pi, |x, y_1\rangle \rangle$$

Here, the phase term can be written as $(y_0y_1 + y_0z + xy_1 + xz - 2xy_0y_1) \cdot \pi$, which expands as a monomials sum of 5 monomial terms, with the following definition and constraints:

Monomial definition	Boolean constraint	Real Coefficient
$\text{Mon}_1 := y_0y_1\pi$	$\text{bc}_1(\mathbf{x}, \mathbf{y}) := y_0 \wedge y_1$	$k_1 := \pi$
$\text{Mon}_2 := y_0z\pi$	$\text{bc}_2(\mathbf{x}, \mathbf{y}) := y_0 \wedge z$	$k_2 := \pi$
$\text{Mon}_3 := xy_1\pi$	$\text{bc}_3(\mathbf{x}, \mathbf{y}) := x \wedge y_1$	$k_3 := \pi$
$\text{Mon}_4 := xz\pi$	$\text{bc}_4(\mathbf{x}, \mathbf{y}) := x \wedge z$	$k_4 := \pi$
$\text{Mon}_5 := -2xy_0y_1\pi$	$\text{bc}_5(\mathbf{x}, \mathbf{y}) := x \wedge y_0 \wedge y_1$	$k_5 := -2\pi$

$\square$

To bring this into WMC form, we proceed monomial by monomial. For each $j \in J$ we introduce a fresh Boolean variable s_j , referred to as a *monomial variable*, together with a constraint

$$\gamma_j : s_j \leftrightarrow \mathbf{bc}_j(\mathbf{x}, \mathbf{y}).$$

Thus, s_j explicitly represents whether the Boolean condition $\mathbf{bc}_j(\mathbf{x}, \mathbf{y})$ holds. We then attach to s_j a literal weight function

$$W(s_j, 0) = 1, \quad W(s_j, 1) = e^{ik_j}.$$

Intuitively, if the condition is false, the term contributes nothing, while if it is true, we multiply by the corresponding root of unity. We set the weight of all other literals to 1, as they are irrelevant to the sum of diagonal amplitude values.

From

$$PS(C)|\mathbf{x}\rangle = \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y}} e^{2\pi i \Phi(\mathbf{x}, \mathbf{y})} |\mathcal{O}(\mathbf{x}, \mathbf{y})\rangle,$$

we obtain

$$\langle \mathbf{x} | PS(C) | \mathbf{x} \rangle = \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y}} e^{2\pi i \Phi(\mathbf{x}, \mathbf{y})} \langle \mathbf{x} | \mathcal{O}(\mathbf{x}, \mathbf{y}) \rangle = \frac{1}{\sqrt{2^m}} \sum_{\mathbf{y}: \mathcal{O}(\mathbf{x}, \mathbf{y}) = \mathbf{x}} e^{2\pi i \Phi(\mathbf{x}, \mathbf{y})}.$$

Here, as observed in [3, Lemma 4.1], we have that by orthonormality $\langle \mathbf{x} | \mathcal{O}(\mathbf{x}, \mathbf{y}) \rangle$ equals the Kronecker delta $\delta_{\mathcal{O}(\mathbf{x}, \mathbf{y}), \mathbf{x}}$, equals 1 if $x = \mathcal{O}(\mathbf{x}, \mathbf{y})$ and 0 otherwise. Hence, in the WMC encoding, we include the formula

$$\beta : \bigwedge_{i=1}^n (\mathcal{O}(\mathbf{x}, \mathbf{y})_i \leftrightarrow x_i),$$

so that only diagonal paths are counted. Finally, we form the global Boolean constraint

$$\Gamma := \beta \wedge \bigwedge_{j \in J} \gamma_j,$$

which enforces both the path restriction and the link between each s_j and its underlying condition.

With this construction, the sum of diagonal amplitude is obtained as

$$\sum_{\mathbf{x} \in \{0,1\}^n} \langle \mathbf{x} | PS(C) | \mathbf{x} \rangle = \frac{1}{\sqrt{2^m}} \text{WMC}(\Gamma, W),$$

where $\text{WMC}(\Gamma, W)$ denotes the weighted model count of Γ under the literal weights W .

Example 6. The equivalence instance in Example 5 can be encoded as a weighted model counting (WMC) problem. Using notations from Example 5:

$$\Gamma = \bigwedge_{j=1}^5 (s_j \leftrightarrow \mathbf{bc}_j) \wedge (y_1 \leftrightarrow z).$$

Since $k_j = \pi$ for all $j \in \{1, 2, 3, 4\}$ and $k_5 = -2\pi$, the weight function W is

$$W(s_j, 0) = 1, \quad W(s_j, 1) = -1^{\delta(j,5)}$$

From the exhaustive enumeration in Table 1 (where shaded rows are the satisfying assignments), we obtain

$$\text{WMC}(\Gamma, W) = \sum_{\tau \models \Gamma} \prod_{j=1}^5 W(s_j, \tau(s_j)) = 8.$$

Since $m = 2$ and $n = 2$,

$$\sum_{\mathbf{x} \in \{0,1\}^2} \langle \mathbf{x} | PS(C) | \mathbf{x} \rangle = 4 = \frac{1}{\sqrt{2^2}} \text{WMC}(\Gamma, W).$$

Therefore, the circuit C_1 is equivalent to the circuit C_2 . Note that WMC solvers normally do not perform such exhaustive enumeration, we include it here to help the reader verify the correctness of the results. $\square$

$x \ z \ y_0 \ y_1$	$s_1 \ s_2 \ s_3 \ s_4 \ s_5$	$\prod W(v, \tau(v))$		$x \ z \ y_0 \ y_1$	$s_1 \ s_2 \ s_3 \ s_4 \ s_5$	$\prod W(v, \tau(v))$
0 0 0 0	0 0 0 0 0	1		1 0 0 0	0 0 0 0 0	1
0 0 0 1	0 0 0 0 0	1		1 0 0 1	0 0 1 0 0	-1
0 0 1 0	0 0 0 0 0	1		1 0 1 0	0 0 0 0 0	1
0 0 1 1	1 0 0 0 0	-1		1 0 1 1	1 0 1 0 1	1
0 1 0 0	0 0 0 0 0	1		1 1 0 0	0 0 0 1 0	-1
0 1 0 1	0 0 0 0 0	1		1 1 0 1	0 0 1 1 0	1
0 1 1 0	0 1 0 0 0	-1		1 1 1 0	0 1 0 1 0	1
0 1 1 1	1 1 0 0 0	1		1 1 1 1	1 1 1 1 1	1

Table 1: Weight computation for the CZ case study

4 Implementation

We implemented the proposed approach in a Python-based tool named QUPRS (Quantum Path-sum Reduction and WMC Solver). For the implementation of the rule-based reduction, we use SYMENGINE [29] and SYMPY [33] to store and manipulate the phase polynomial part and Boolean function part of the path-sum formalism. For solving weighted model counting problems, we integrated GPMC [39], which was selected due to its support for negative weights and good performance in the Model Counting Competition.¹⁰

¹⁰ To meet the specific needs of our framework, we use an extended version of GPMC that handles complex-valued weights, available at <https://github.com/System-Verification-Lab/GPMC>.

benchmarks				Hybrid		RR		WMC	
Benchmark Name	q	G1	G2	time	result	time	result	time	result
ghz	32	34	33	0.027	△	0.027	△	0.08	△
	64	66	65	0.084	△	0.084	△	0.212	△
	128	130	129	0.306	△	0.306	△	0.617	△
graphstate	16	160	108	0.162	○	0.162	○	0.286	○
	32	320	219	0.33	○	0.33	○	0.778	△
	64	640	435	1.068	△	1.068	△	2.427	△
grover	4	190	160	0.213	△	0.162	×	0.446	△
	5	499	409	0.194	△	0.194	△	0.743	△
	6	1568	1400	>200	TO	>200	TO	28.579	△
qaoa	7	133	170	1.023	△	0.915	×	1.031	△
	9	171	209	0.974	△	0.793	×	0.586	△
	11	209	270	1.636	△	1.442	×	1.274	△
qft	16	672	543	1.818	△	0.738	×	1.932	△
	20	1040	825	20.122	△	1.445	×	20.706	△
	22	1254	970	82.866	△	2.004	×	83.452	△
	24	1488	1111	>200	TO	2.497	×	>200	TO
qnn	4	111	122	0.738	△	0.645	×	0.573	△
	8	319	334	15.694	△	1.142	×	32.203	△
	12	623	648	>200	TO	4.624	×	>200	TO
qpeexact	8	192	150	0.326	△	0.326	△	0.355	△
	16	712	562	0.33	△	0.33	△	37.901	△
	20	1092	861	0.581	△	0.58	×	>200	TO
	24	1537	1173	>200	TO	2.219	×	>200	TO
qpeinexact	8	192	150	0.333	△	0.333	△	0.608	△
	16	712	562	0.295	△	0.295	△	32.205	△
	20	1092	861	0.623	△	0.62	×	>200	TO
	24	1552	1175	1.075	△	1.049	×	>200	TO
qwalk	3	141	113	0.107	○	0.107	○	0.635	○
	4	357	294	0.154	△	0.154	△	0.677	△
	5	1305	1158	>200	TO	>200	TO	18.207	△
vqe	4	66	44	0.388	△	0.26	×	0.19	△
	8	134	95	0.818	△	0.41	×	0.216	△
	16	270	211	2.878	○	2.335	×	0.598	○
wstate	4	55	25	0.147	△	0.147	△	0.116	△
	8	127	61	0.182	△	0.128	×	0.345	△
	16	271	129	TO	×	11.029	×	TO	×

Table 2: Comparison of Operation Modes.

Our tool is publicly available under the MIT license. All experiments were conducted on an Azure Standard E8ads v5 server (8 vCPUs, 64 GiB memory).

The QuPRS tool supports three operation modes. Assume that we are checking the equivalence of two circuits C_1 and C_2 , all modes first construct the path-sum formula of $C_2(C_1)^\dagger$:

- **RR (Reduction-rules)** Applies only rule-based reductions during the path-sum formula construction. If the resulting formula differs from that of an identity matrix, the tool reports “unknown.”
- **WMC (Weighted Model Counting)** Constructs the path-sum formula without applying any reduction rules. It uses weighted model counting to determine if the final path-sum formula corresponds to the identity.
- **Hybrid** Initially applies reduction rules to simplify the formula. If the resulting formula is not syntactically identical to that of the identity, weighted model counting is employed to check for semantic equivalence.

5 Experimental Evaluation

To evaluate the performance of our proposed approach, we designed three sets of experiments.

5.1 Comparison of Operation Modes

In the first experiment, we compare the performance of the three operation modes (RR, WMC, and Hybrid) of QUPRS. We use the benchmark suite from [31], which includes quantum algorithm circuits such as GHZ, QAOA, VQE, QNN, and Grover, among others. The original circuits are expressed using the gate set Clifford+T, Rx, Ry, Rz. To facilitate comparison, we use QISKIT [26] (version 1.4.2) to transpile these circuits to the gate set {H, Ry, Rz, CNOT}, and perform equivalence checking between the original and transpiled ones.

The details are presented in Table 2. In the table, q denotes the number of qubits in the benchmark circuits, while G_1 and G_2 represent the total number of gates under the two different gate sets, respectively. For each operation mode, we record the runtime for equivalence checking and the corresponding result. For the **result**, the symbols $\bigcirc$ and Δ indicate that the two circuits are equivalent or equivalent up to a global phase, respectively, whereas “TO” denotes a timeout where the runtime exceeds 200 seconds. The symbol $\times$ indicates that the tool fails to verify equivalence. We highlight the best-performing mode in for each example.

Overall, the results align with our expectations: the **Hybrid** mode consistently outperforms the other two, particularly on more challenging benchmarks. Specifically, **RR** alone solves only 14 out of the 35 examples, while **WMC** solves 28, and **Hybrid** solves 29. Examining the solving time for the largest instance within each algorithm category, we observe that **Hybrid** performs better for GHZ, GraphState, QFT, QNN, QPE-Exact, QPE-Inexact, QWalk, and WState constructions. In contrast, **WMC** outperforms Hybrid in only three categories: Grover, QAOA, and VQE benchmarks.

Upon further investigation, we found that the inefficiency in the three categories (Grover, QAOA, and VQE) where **WMC** outperforms **Hybrid** stems from a low *hit rate*. Specifically, the percentage of instances where the path-sum formula matches the patterns required for applying a reduction rule is typically low, around 50%. This reduced hit rate significantly limits the effectiveness of rule-based simplifications, making pure WMC more efficient in these cases.

5.2 Comparison with Other RR and WMC Implementations

One might argue that the superior performance of the **Hybrid** mode could stem from an inefficient implementation of the **RR** or **WMC** modes within QUPRS. To address this concern, we further compare QUPRS against state-of-the-art tools: FEYMANN [21], which implements the path-sum reduction approach, and QUOKKA# [31], which implements a WMC-based equivalence checking strategy, each using their own benchmark suites¹¹. We want to again emphasize that although both QUOKKA# and **WMC** reduced the circuit equivalence problem

¹¹ We transpiled the Feynman benchmark to the basis gates [H, Y, Z, T, $T^\dagger$, CX] using Qiskit 1.2.4. We encountered transpilation issues with Qiskit 1.4.2 on this set of benchmarks. All transpiled files are available on our GitHub repository.

to weighted model counting, they use very different encodings. We still compare the two because they are already the closest.

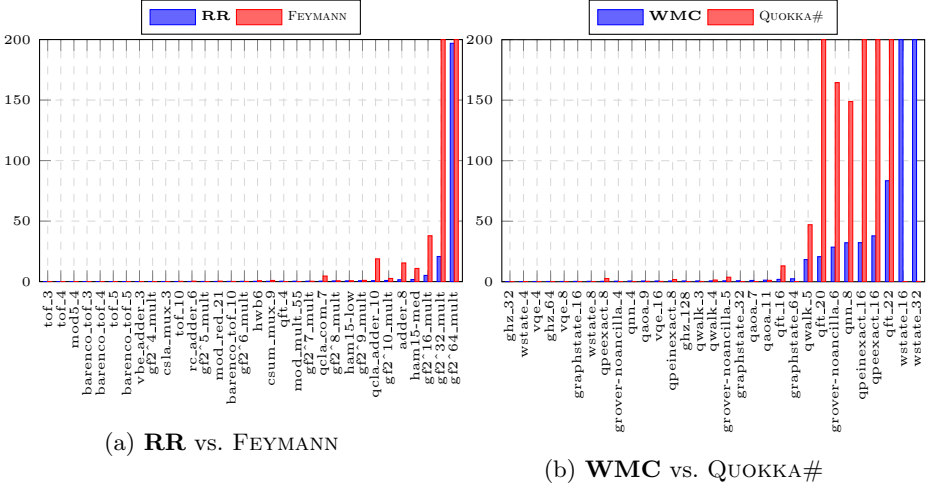


Fig. 3: Comparisons between QUPRS and other tools.

In Fig. 3a, we present a bar chart comparing the **RR** mode of QUPRS with FEYMANN. The x-axis represents the benchmark example names, while the y-axis indicates the run time. A timeout of 80 seconds is enforced. The results demonstrate that the **RR** mode of QUPRS consistently outperforms FEYMANN on more challenging examples, specifically, those requiring more than one second to solve.

We compare between the **WMC** mode of QUPRS and QUOKKA# in Fig. 3b. As before, the x-axis represents the benchmark example names, and the y-axis indicates the run time, with a timeout of 200 seconds enforced. The results show that the performance of the two tools is comparable. QUOKKA# outperforms QUPRS on four examples: graphstate 32, graphstate 64, wstate 16, and wstate 32. This outcome is somewhat expected. Although both tools use a weighted model counter as their backend engine, they adopt very different encodings of the path formula. The performance of model counters is highly sensitive to the encoding; even for the same underlying concept, different encodings can lead to vastly different execution times. Therefore, it is unsurprising that QUPRS performs better in some cases, while QUOKKA# excels in others. Overall, we believe that it is fair to conclude that our implementation is comparable to the state-of-the-art implementation of the same algorithm.

5.3 Comparison with Other Categories of Equivalence Checkers

Finally, we evaluate QUPRS against a broader selection of state-of-the-art quantum circuit equivalence checking tools, spanning various methodological categories. We selected the following tools for comparison. We highlight the best-performing tool in ■ and second performing tool in ■ for each example:

benchmarks					QUOKKA#					QCEC					PYZX				
Name	q	G	C2	time result	time result	time result	time result	time result	time result	time result	time result	time result	time result	time result	time result	time result	time result	time result	time result
ghz	32	34	33	0.027	Δ	0.017	Δ	0.03	Δ	0.014	Δ	0.014	Δ	0.014	Δ	0.014	Δ	0.014	Δ
	64	66	65	0.084	Δ	0.019	Δ	0.035	Δ	0.024	Δ	0.024	Δ	0.024	Δ	0.024	Δ	0.024	Δ
	128	130	129	0.306	Δ	0.023	Δ	0.048	Δ	0.055	Δ	0.055	Δ	0.055	Δ	0.055	Δ	0.055	Δ
	16	160	108	0.162	○	0.032	Δ	0.027	Δ	0.026	×	0.026	×	0.026	×	0.026	×	0.026	×
graphstate	32	320	219	0.33	○	0.05	Δ	0.031	Δ	0.045	×	0.045	×	0.045	×	0.045	×	0.045	×
	64	640	435	1.068	Δ	0.113	Δ	0.044	Δ	0.143	×	0.143	×	0.143	×	0.143	×	0.143	×
	4	190	160	0.213	Δ	0.092	Δ	0.026	Δ	0.056	×	0.056	×	0.056	×	0.056	×	0.056	×
grover	5	499	409	0.194	Δ	3.719	Δ	0.03	Δ	0.458	×	0.458	×	0.458	×	0.458	×	0.458	×
	6	1568	1400	>200	TO	164.546	Δ	0.076	Δ	4.734	×	4.734	×	4.734	×	4.734	×	4.734	×
	7	133	170	1.023	Δ	0.069	Δ	0.033	○	0.043	×	0.043	×	0.043	×	0.043	×	0.043	×
qaoa	9	171	209	0.974	Δ	0.322	Δ	0.056	Δ	0.122	×	0.122	×	0.122	×	0.122	×	0.122	×
	11	209	270	1.636	Δ	1.142	Δ	0.065	Δ	0.141	×	0.141	×	0.141	×	0.141	×	0.141	×
	16	672	543	1.818	Δ	13.107	Δ	0.712	×	0.391	×	0.391	×	0.391	×	0.391	×	0.391	×
qft	20	1040	825	20.122	Δ	>200	TO	18.602	×	0.693	×	0.693	×	0.693	×	0.693	×	0.693	×
	22	1254	970	82.966	Δ	>200	TO	157.183	×	0.934	×	0.934	×	0.934	×	0.934	×	0.934	×
	24	1488	1111	>200	TO	>200	TO	>200	TO	1.297	×	1.297	×	1.297	×	1.297	×	1.297	×
	4	111	122	0.738	Δ	0.015	Δ	0.031	Δ	0.047	×	0.047	×	0.047	×	0.047	×	0.047	×
qnn	8	319	334	15.694	Δ	148.927	Δ	0.041	Δ	0.172	×	0.172	×	0.172	×	0.172	×	0.172	×
	12	623	648	>200	TO	>200	TO	0.173	Δ	1.412	×	1.412	×	1.412	×	1.412	×	1.412	×
	8	192	150	0.326	Δ	2.645	Δ	0.042	Δ	0.126	×	0.126	×	0.126	×	0.126	×	0.126	×
qpeexact	16	712	562	0.33	Δ	>200	TO	>200	TO	1.978	×	1.978	×	1.978	×	1.978	×	1.978	×
	20	1092	861	0.589	Δ	>200	TO	>200	TO	4.612	×	4.612	×	4.612	×	4.612	×	4.612	×
	24	1537	1173	>200	TO	>200	TO	>200	TO	7.145	×	7.145	×	7.145	×	7.145	×	7.145	×
qpeinexact	8	192	150	0.333	Δ	1.82	Δ	0.101	Δ	0.167	Δ	0.167	Δ	0.167	Δ	0.167	Δ	0.167	Δ
	16	712	562	0.295	Δ	>200	TO	>200	TO	2.853	Δ	2.853	Δ	2.853	Δ	2.853	Δ	2.853	Δ
	20	1092	861	0.629	Δ	>200	TO	>200	TO	8.937	×	8.937	×	8.937	×	8.937	×	8.937	×
	24	1552	1175	1.075	Δ	>200	TO	>200	TO	18.478	×	18.478	×	18.478	×	18.478	×	18.478	×
qwalk	3	141	113	0.107	○	0.19	Δ	0.096	○	0.179	×	0.179	×	0.179	×	0.179	×	0.179	×
	4	357	294	0.154	Δ	1.399	Δ	0.121	Δ	1.256	×	1.256	×	1.256	×	1.256	×	1.256	×
	5	1305	1158	>200	TO	47.09	Δ	0.225	Δ	5.025	×	5.025	×	5.025	×	5.025	×	5.025	×
vqe	4	66	44	0.388	Δ	0.095	Δ	0.109	Δ	0.083	×	0.083	×	0.083	×	0.083	×	0.083	×
	8	134	95	0.818	Δ	0.135	Δ	0.125	○	0.116	×	0.116	×	0.116	×	0.116	×	0.116	×
	16	270	211	2.878	○	0.258	Δ	>200	TO	0.276	×	0.276	×	0.276	×	0.276	×	0.276	×
wstate	4	55	25	0.147	Δ	0.079	Δ	0.102	Δ	0.072	×	0.072	×	0.072	×	0.072	×	0.072	×
	8	127	61	0.182	Δ	0.111	Δ	0.094	Δ	0.092	×	0.092	×	0.092	×	0.092	×	0.092	×
	16	271	129	>200	TO	0.123	Δ	0.11	Δ	0.083	×	0.083	×	0.083	×	0.083	×	0.083	×
	32	559	261	>200	TO	0.256	Δ	0.114	Δ	0.407	×	0.407	×	0.407	×	0.407	×	0.407	×

(a) Normal case.

(b) Random rotation gate injected.

Table 3: Comparison with State-of-the-art Tools

- QUOKKA#, a tool based on weighted model counting (WMC)
- QCEC [36], which combines three techniques ZX-calculus, QMDDs, and simulation, executed in parallel
- FEYMANN, which relies on circuit reduction rules
- SLIQEC [43], a tool based on binary decision diagrams (BDD)
- PYZX [28], which uses ZX-calculus
- TDD [25][23][24], which employs tensor decision diagrams (TDD)

We found that both FEYMANN and SLIQEC do not support rotation gates such as RX, RY, and RZ, and therefore are unable to handle the majority of our benchmark examples. TDD frequently produces incorrect results, likely due to floating-point precision issues. Consequently, we only report detailed comparative results for QUOKKA#, QCEC, and PYZX in Table 3, using the same format as in Table 2. Here QUPRS means the Hybrid model of the tool. The timeout for each tool is set to 200 seconds.

From Table 3 (a), we observe that the three tools QUPRS, QUOKKA#, and QCEC are complementary. QUPRS achieves the best results on the quantum Fourier transform and phase estimation algorithms, which involve many distinct rotation angles and require precise numerical reasoning about phase accumulation. QUOKKA# performs better on the GHZ and variational quantum algorithms, while QCEC excels on graph-state preparation, Grover search, QAOA, quantum neural network, W-state, and quantum random walk benchmarks.

QUPRS is particularly effective for circuits whose unitary transformations depend on continuous parameters such as rotation angles or phase shifts, where verifying equivalence requires reasoning about exact numerical values. In contrast, algorithms such as Grover’s search rely mainly on discrete structural symmetries and repetitive iterations, which are better handled by combinatorial reasoning as in QCEC. A unified framework combining the complementary strengths of all three tools would yield a state-of-the-art virtual equivalence checker.

In Table 3(b), we evaluate the robustness of the tools by injecting a random rotation gate into each circuit to simulate a phase error and check whether the tools can detect it. A $\times$ indicates that the injected error is successfully detected. For reference, we also include the WMC configuration of QUPRS as a separate column. PYZX is omitted because it fails to detect any of the injected errors.

QUPRS detects the most errors, 28 out of 37 cases, demonstrating strong sensitivity to subtle numerical discrepancies. Its WMC mode detects the same number of buggy circuits but in different instances, suggesting complementary reasoning mechanisms within the framework. In comparison, QUOKKA# detects 27 and QCEC 24 cases. These results highlight the superior capability of QUPRS in identifying small yet meaningful deviations in circuit behavior.

6 Conclusion

We presented a hybrid verifier for quantum circuit equivalence that combines path-sum reductions with a new weighted model counting (WMC) encoding. The reductions aggressively simplify circuits when algebraic structure is present, and the WMC back end provides a complete semantic check up to global phase on the residual path-sum. We implemented the approach in QUPRS, which supports reduction-only, WMC-only, and hybrid modes. On standard benchmarks, the hybrid mode consistently outperforms either component alone and is competitive with state-of-the-art tools. It is especially effective on QFT and phase estimation, where correctness depends on many distinct rotation angles and precise phase accumulation. The robustness study with injected random rotations shows the highest error-detection rate among the compared tools. Overall, QUPRS demonstrates that coupling symbolic reductions with complete model-based reasoning yields a practical and scalable path to quantum circuit equivalence checking.

Acknowledgments This work was supported by National Science and Technology Council, R.O.C., projects NSTC 114-2221-E-027-044 -MY2 and NSTC 114-2119-M-001-002-; Air Force Office of Scientific Research project FA2386-23-1-4107; Academia Sinica Investigator Project Grant AS-IV-114-M07; Foxconn project; French National Research Agency (ANR) HQI project ANR-22-PNCQ0001; Quantum Delta NL (QDNL) project.

Data Availability All data and code supporting the findings of this study are openly available in the repository <https://github.com/PhysicsQoo/pathsum>. The repository contains the implementation, example inputs, and artifacts required to reproduce the experimental results reported in this paper.

References

1. Parosh Aziz Abdulla, Yo-Ga Chen, Yu-Fang Chen, Lukáš Holík, Ondřej Lengál, Jyun-Ao Lin, Fang-Yi Lo, and Wei-Lun Tsai. Verifying quantum circuits with level-synchronized tree automata. *Proc. ACM Program. Lang.*, 9(POPL), January 2025.
2. Rajeev Acharya, Laleh Aghababaie-Beni, Igor Aleiner, Trond I Andersen, Markus Ansmann, Frank Arute, Kunal Arya, Abraham Asfaw, Nikita Astrakhantsev, Juan Atalaya, et al. Quantum error correction below the surface code threshold. *arXiv preprint arXiv:2408.13687*, 2024.
3. Matthew Amy. Towards large-scale functional verification of universal quantum circuits. *arXiv preprint arXiv:1805.06908*, 2018.
4. Matthew Amy. Complete equational theories for the sum-over-paths with unbalanced amplitudes. *Electronic Proceedings in Theoretical Computer Science*, 384:127–141, August 2023.
5. Matthew Amy and Joseph Lunderville. Linear and non-linear relational analyses for quantum program optimization. *Proceedings of the ACM on Programming Languages*, 9(POPL):1072–1103, 2025.
6. Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.
7. Fabian Bauer-Marquart, Stefan Leue, and Christian Schilling. symQV: Automated symbolic verification of Quantum Programs. In Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker, editors, *Formal Methods*, pages 181–198, Cham, 2023. Springer International Publishing.
8. Lukas Burgholzer and Robert Wille. Advanced equivalence checking for quantum circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(9):1810–1824, 2020.
9. Christophe Charetton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. An automated deductive verification framework for circuit-building quantum programs. In *European Symposium on Programming*, pages 148–177. Springer International Publishing Cham, 2021.
10. Tian-Fu Chen, Jie-Hong R. Jiang, and Min-Hsiu Hsieh. Partial equivalence checking of quantum circuits. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 594–604, 2022.
11. Yu-Fang Chen, Kai-Min Chung, Min-Hsiu Hsieh, Wei-Jia Huang, Ondřej Lengál, Jyun-Ao Lin, and Wei-Lun Tsai. Autoq 2.0: From verification of quantum circuits to verification of quantum programs. In *TACAS 2025*, pages 87–108. Springer, 2025.
12. Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, and Wei-Lun Tsai. AutoQ: An automata-based quantum circuit verifier. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III*, volume 13966 of *Lecture Notes in Computer Science*, pages 139–153. Springer, 2023.
13. Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. An automata-based framework for verification and bug hunting in quantum circuits. *Proc. ACM Program. Lang.*, 7(PLDI):1218–1243, 2023.
14. Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. An automata-based framework for verification and bug hunting in quantum circuits. *Commun. ACM*, 68(6):85–93, June 2025.

15. Yu-Fang Chen, Philipp Rümmer, and Wei-Lun Tsai. A theory of cartesian arrays (with applications in quantum circuit verification). In *International Conference on Automated Deduction*, pages 170–189. Springer, 2023.
16. Alexandre Clément, Nicolas Heurtel, Shane Mansfield, Simon Perdrix, and Benoît Valiron. Lov-calculus: A graphical language for linear optical quantum circuits. *arXiv preprint arXiv:2204.11787*, 2022.
17. Bob Coecke and Ross Duncan. Interacting quantum observables: Categorical algebra and diagrammatics. *New Journal of Physics*, 13(4):043016, Apr 2011.
18. Bob Coecke and Aleks Kissinger. *Picturing quantum processes*. Cambridge University Press, Cambridge, United Kingdom, 2017.
19. Ross Duncan and Simon Perdrix. Graph states and the necessity of euler decomposition. In *Conference on Computability in Europe*, pages 167–177. Springer, 2009.
20. Ross Duncan and Simon Perdrix. Rewriting measurement-based quantum computations with generalised flow. In *International Colloquium on Automata, Languages, and Programming*, pages 285–296. Springer, 2010.
21. Richard P Feynman, Albert R Hibbs, and Daniel F Styer. *Quantum mechanics and path integrals*. Courier Corporation, 2010.
22. Daniel Gottesman. The heisenberg representation of quantum computers. *arXiv preprint quant-ph/9807006*, 1998.
23. Xin Hong, Wei-Jia Huang, Wei-Chen Chien, Yuan Feng, Min-Hsiu Hsieh, Sanjiang Li, Chia-Shun Yeh, and Mingsheng Ying. Decision diagrams for symbolic verification of quantum circuits. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 1, pages 970–977. IEEE, 2023.
24. Xin Hong, Wei-Jia Huang, Wei-Chen Chien, Yuan Feng, Min-Hsiu Hsieh, Sanjiang Li, and Mingsheng Ying. Equivalence checking of parameterised quantum circuits. *arXiv preprint arXiv:2404.18456*, 2024.
25. Xin Hong, Xiangzhen Zhou, Sanjiang Li, Yuan Feng, and Mingsheng Ying. A tensor network based decision diagram for representation of quantum circuits. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 27(6):1–30, 2022.
26. Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with Qiskit, 2024.
27. Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. A complete axiomatisation of the zx-calculus for clifford+t quantum mechanics. LICS '18, page 559–568, New York, NY, USA, 2018. Association for Computing Machinery.
28. Aleks Kissinger and John van de Wetering. PyZX: Large Scale Automated Diagrammatic Reasoning. In Bob Coecke and Matthew Leifer, editors, *Proceedings 16th International Conference on Quantum Physics and Logic*, Chapman University, Orange, CA, USA., 10-14 June 2019, volume 318 of *Electronic Proceedings in Theoretical Computer Science*, pages 229–241. Open Publishing Association, 2020.
29. Jialin Ma, Isuru Fernando, and Xin Chen. *symengine: Interface to the 'SymEngine' Library*, 2022. R package version 0.2.2.
30. Jingyi Mei, Marcello Bonsangue, and Alfons Laarman. Simulating quantum circuits by model counting. In *CAV 2024, (accepted for publication)*. Springer, 2024. Preprint available at arXiv:2403.07197.
31. Jingyi Mei, Tim Coopmans, Marcello Bonsangue, and Alfons Laarman. Equivalence checking of quantum circuits by model counting. In *International Joint Conference on Automated Reasoning*, pages 401–421. Springer, 2024.

32. Jingyi Mei, Jan Martens, and Alfons Laarman. Disentangling the gap between quantum and $\#$ sat. In *International Colloquium on Theoretical Aspects of Computing*, pages 17–40. Springer, 2024.
33. Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017.
34. Michael A. Nielsen and Isaac Chuang. *Quantum computation and quantum information*. Cambridge University Press, Cambridge, United Kingdom, 2002.
35. Tom Peham, Lukas Burgholzer, and Robert Wille. Equivalence checking of parameterized quantum circuits: Verifying the compilation of variational quantum algorithms. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference, ASPDAC '23*, page 702–708, New York, NY, USA, 2023. Association for Computing Machinery.
36. Nils Quetschlich, Lukas Burgholzer, and Robert Wille. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum*, 2023. MQT Bench is available at <https://www.cda.cit.tum.de/mqtbench/>.
37. Jérôme Ricciardi, Sébastien Bardin, Christophe Chareton, and Benoît Valiron. Quantum circuit equivalence checking: A tractable bridge from unitary to hybrid circuits. In *3rd International Workshop on Services and Quantum Software*, 2025.
38. Meghana Sistla, Swarat Chaudhuri, and Thomas Reps. Symbolic quantum simulation with quasimodo. In *International Conference on Computer Aided Verification*, pages 213–225. Springer, 2023.
39. Ryosuke Suzuki, Kenji Hashimoto, and Masahiko Sakai. Improvement of projected model-counting solver with component decomposition using sat solving in components. Technical report, Technical report, JSAI Technical Report, SIG-FPAI-103-B506, 2017.
40. Yu Tanaka. Exact non-identity check is nqp-complete. *International Journal of Quantum Information*, 8(05):807–819, 2010.
41. Renaud Vilmart. The structure of sum-over-paths, its consequences, and completeness for clifford. In *Foundations of Software Science and Computation Structures*, volume 12650, page 531, 2021.
42. Renaud Vilmart. Rewriting and completeness of sum-over-paths in dyadic fragments of quantum computing. *Logical Methods in Computer Science*, Volume 20, Issue 1, March 2024.
43. Chun-Yu Wei, Yuan-Hung Tsai, Chiao-Shan Jhang, and Jie-Hong R Jiang. Accurate bdd-based unitary operator manipulation for scalable and robust quantum circuit verification. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 523–528, 2022.

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Error-Tolerant Quantum State Discrimination: Optimization and Quantum Circuit Synthesis

Chien-Kai Ma[✉], Bo-Hung Chen[✉], Tian-Fu Chen[✉], Dah-Wei Chiou[✉], and
Jie-Hong R. Jiang[✉]

Graduate Institute of Electronics Engineering, National Taiwan University, Taiwan

Abstract. We develop *error-tolerant* quantum state discrimination (QSD) strategies that maintain reliable performance under moderate noise. Two complementary approaches are proposed: *CrossQSD*, which generalizes unambiguous discrimination with tunable confidence bounds to balance accuracy and efficiency, and *FitQSD*, which optimizes the measurement outcome distribution to approximate that of the ideal noiseless case. Furthermore, we provide a unified *hybrid-objective* QSD framework that continuously interpolates between minimum-error discrimination (MED) and FitQSD, allowing flexible trade-offs among competing objectives. The associated optimization problems are formulated as convex programs and efficiently solved via disciplined convex programming or, in many cases, semidefinite programming. Additionally, a circuit synthesis framework based on a modified Naimark dilation and isometry synthesis enables hardware-efficient implementations with substantially reduced qubit and gate resources. An open-source toolkit automates the full optimization and synthesis workflow, providing a practical route to QSD on current quantum devices.

1 Introduction

Quantum state discrimination (QSD) [8,6,10] is a fundamental problem in quantum information science that concerns how to identify, as accurately and efficiently as possible, the state of a quantum system known to be prepared in one of several predefined states. This task underpins many areas of quantum technology, including quantum cryptography, quantum communication, and quantum computing [9,26]. Unlike classical states, quantum states can be nonorthogonal and therefore cannot be perfectly distinguished by any physical measurement. This intrinsic indistinguishability is not a technological limitation but a fundamental feature of quantum mechanics. It plays a pivotal role in the security of quantum key distribution protocols such as BB84 [9] and SARG04 [26], where information is encoded in nonorthogonal states so that any eavesdropping attempt inevitably disturbs the system and can thus be detected.

Over the years, several strategies have been developed to address QSD, each offering a distinct trade-off between accuracy, confidence, and efficiency. The most widely studied approaches are *minimum-error discrimination* (MED) [8,25], which minimizes the average probability of incorrectly identifying a state, and *unambiguous quantum state discrimination* (UQSD) [10,12], which permits

inconclusive outcomes to ensure that every conclusive result is error-free. UQSD, for instance, plays a key role in the B92 protocol [20], where each received state must be either reliably identified or discarded. Beyond these, other strategies pursue different optimization objectives. *Maximum-confidence discrimination* [13] aims to maximize the confidence that a conclusive outcome correctly identifies the state, even at the cost of allowing occasional errors or inconclusive results, while *worst-case a posteriori probability discrimination* [21] seeks to maximize the worst-case posterior probability of being correct, ensuring robust performance in the least favorable cases. Together, these approaches illustrate the diverse operational perspectives and optimization criteria that underlie the general problem of quantum state discrimination.

In realistic settings, however, unavoidable noise introduces additional errors and can significantly undermine the performance of conventional strategies, which are typically formulated under ideal, noiseless assumptions. For example, when depolarizing noise affects all predefined states, no matter how small the noise is, UQSD will always yield an inconclusive outcome, rendering the method ineffectual. To address such challenges, the *fixed rate of inconclusive outcome* (FRIO) method [7,17] has been proposed as a compromise between MED and UQSD: it relaxes the guarantee of conclusive correctness while enforcing an upper bound on the rate of inconclusive outcomes.

In this work, we further tackle the issue of noise from multiple perspectives by developing *noise-aware* approaches that maintain reliable performance under realistic noisy conditions. Specifically, we propose two novel error-tolerant QSD strategies, along with some auxiliary methods that account for different aspects of error tolerance. The first strategy, *CrossQSD*, generalizes UQSD by incorporating tunable confidence bounds associated with false positive and false negative errors,¹ thereby enabling a more controlled balance between accuracy, confidence, and efficiency. The second strategy, *FitQSD*, optimizes the probability distribution of measurement outcomes to reproduce, as closely as possible, that of the ideal noiseless UQSD case, thus preserving discrimination performance even in the presence of noise. Furthermore, MED and FitQSD can be integrated into a unified *hybrid-objective QSD* framework, offering a continuous interpolation between MED and UQSD and enabling a flexible trade-off among multiple optimization objectives under noisy conditions.

The optimization problems underlying both CrossQSD and FitQSD, as well as all auxiliary and hybrid methods, can be elegantly expressed within the general framework of convex optimization [11,5] and efficiently solved using disciplined convex programming (DCP) [16] or its parametric extension, disciplined parametrized programming (DPP) [1]. In many instances, these formulations further reduce to semidefinite programming (SDP) [27,14], which allows even more efficient numerical solutions. This formulation naturally yields the optimal

¹ A false negative with respect to a specific predefined state ρ_i refers to the event that the given state is ρ_i but it is not identified as ρ_i , i.e., the outcome corresponds to ρ_j with $j \neq i$. Conversely, a false positive for ρ_i refers to the event that the outcome identifies the state as ρ_i while the given state is actually ρ_j with $j \neq i$.

positive operator-valued measure (POVM) and readily extends to cases involving both mixed and pure predefined states, thereby enabling a systematic investigation of noise effects on discrimination performance and the comparative advantages of different approaches under various conditions.

Even when the optimal POVM for a given QSD strategy is known, its physical realization on quantum hardware is often nontrivial. Although quantum circuit synthesis for POVMs has been studied, a general synthesis flow tailored for QSD problems has been lacking. To bridge this gap, as our second main contribution, we develop a circuit synthesis framework based on a modified version of Naimark's dilation theorem [24] and the state-of-the-art isometry circuit synthesis [18], enabling efficient implementation of QSD strategies on quantum circuits. For specific cases, such as coherent states relevant to optical quantum systems [15,4], we demonstrate that our approach needs fewer qubits and can substantially reduce gate counts and circuit depth, with only a modest trade-off in discrimination success probability.

Although our primary focus is on the discrimination of multi-qubit states within quantum-circuit architectures, the constructed circuits faithfully capture the essential information flow of the measurement process and can be adapted to other physical platforms where basic quantum gates are available. To support practical implementations, we also provide an open-source toolkit that automates the optimization and synthesis of the proposed QSD strategies. With these tools and methodologies, our work establishes a systematic foundation for performing practical QSD on current and emerging quantum hardware.

2 Error-Tolerant Quantum State Discrimination

The problem of QSD can be formulated in its most general form as follows. Given a predefined set of k quantum states $\{\rho_1, \rho_2, \dots, \rho_k\}$ (each being either a pure or mixed state), Alice prepares one of them with prior probabilities $\{p_1, p_2, \dots, p_k\}$ satisfying $\sum_{i=1}^k p_i = 1$, and sends it to Bob through a quantum channel $\mathcal{E} : \rho_i \mapsto \mathcal{E}(\rho_i) = \rho'_i$, which may or may not introduce noise. Upon receiving ρ'_i , Bob performs a POVM measurement characterized by a POVM $\{\Pi_1, \Pi_2, \dots, \Pi_k\}$, where each element Π_i is a positive-definite operator corresponding to the outcome by which Bob guesses the prepared state to be ρ_i . Alternatively, if an inconclusive outcome is allowed, the measurement is extended to $\{\Pi_1, \Pi_2, \dots, \Pi_k, \Pi_{k+1} \equiv \Pi_?\}$, where $\Pi_?$ represents the operator associated with the inconclusive result. The problem of QSD is to solve the optimal POVM, by which Bob can identify the prepared state as accurately and efficiently as possible under various intended conditions or considerations.

Our toolkit supports multiple types of optimization strategies and is readily extensible to any strategy that can be formulated as a convex optimization problem with constraints expressed in terms of POVM operators. In this section, we formulate the CrossQSD and FitQSD methods, together with other modified QSD schemes, each targeting different optimization objectives. We demonstrate the efficacy and robustness of these schemes on predefined sets of nearly orthogo-

nal states and truncated coherent states. For simplicity, noise in our experiments is modeled by a depolarizing channel

$$\mathcal{E}_\lambda(\rho) : \rho \mapsto (1 - \lambda)\rho + \lambda \frac{I}{d}, \quad (1)$$

where λ denotes the noise level and d is the Hilbert-space dimension.

2.1 Modified Minimum Error Discrimination (MED⁺)

We first introduce MED⁺, a straightforward extension of the conventional MED scheme that allows an inconclusive outcome, similar to UQSD. The optimization problem can be cast as a semidefinite program:

$$\begin{aligned} \text{maximize}_{\{\Pi_i\}} \quad & P_{\text{succ}} = \sum_{i=1}^k p_i \text{Tr}(\rho_i \Pi_i), \end{aligned} \quad (2a)$$

$$\text{subject to} \quad \Pi_i \succeq 0, \quad i = 1, \dots, k+1, \quad (2b)$$

$$\sum_{i=1}^{k+1} \Pi_i = I, \quad (2c)$$

where the optimization objective is the “success probability” P_{succ} —the overall probability of correctly identifying the prepared state—and $A \succeq B$ indicates that $A - B$ is positive semidefinite.

Although extending MED to MED⁺ intuitively enlarges the feasible space and appears to enhance optimality, it in fact provides no additional benefit, as established in the following theorem, whose proof is provided in [23].

Theorem 1. *Let both MED and MED⁺ be applied under the same predefined states $\{\rho_1, \rho_2, \dots, \rho_k\}$ and prior probabilities $\{p_1, p_2, \dots, p_k\}$. Then, in the optimal MED⁺ solution, the inconclusive operator necessarily vanishes, i.e., $\Pi_{?} = 0$, rendering the optimal MED⁺ identical to the optimal MED. In particular, the optimal success probabilities of the two schemes are equal, i.e.,*

$$(P_{\text{succ}})^{\text{MED}} = (P_{\text{succ}})^{\text{MED}^+}. \quad (3)$$

Despite yielding no improvement in optimality, the distinction between MED and MED⁺ offers valuable conceptual insight. Specifically, Theorem 1 immediately implies the following corollary, whose proof is provided in [23].

Corollary 1. *For the same predefined states and prior probabilities, the optimal success probability of MED is always greater than or equal to that of UQSD; that is,*

$$(P_{\text{succ}})^{\text{MED}} \geq (P_{\text{succ}})^{\text{UQSD}}. \quad (4)$$

This result highlights a fundamental distinction between the two strategies: MED allows a nonzero probability of error to maximize the overall success rate, whereas UQSD strictly forbids any incorrect identification and therefore achieves a smaller (or equal) success probability. In other words, UQSD prioritizes certainty over completeness, while MED optimizes performance under uncertainty.

2.2 CrossQSD

To enable controlled tolerance of both false positive and false negative errors within user-specified limits, we propose *CrossQSD*, a relaxation of UQSD formulated as a semidefinite program:

$$\underset{\{\Pi_i\}}{\text{maximize}} \quad P_{\text{succ}} = \sum_{i=1}^k p_i \text{Tr}[\mathcal{E}_\lambda(\rho_i)\Pi_i], \quad (5a)$$

$$\text{subject to} \quad \Pi_i \succeq 0, \quad i = 0, \dots, k+1, \quad (5b)$$

$$\sum_{i=0}^{k+1} \Pi_i = I, \quad (5c)$$

$$p(\Pi_i | \rho_i) = \frac{p_i \text{Tr}[\mathcal{E}_\lambda(\rho_i)\Pi_i]}{\sum_{j=1}^k p_j \text{Tr}[\mathcal{E}_\lambda(\rho_j)\Pi_j]} \geq 1 - \alpha_i, \quad i = 1, \dots, k, \quad (5d)$$

$$p(\rho_i | \Pi_i) = \frac{p_i \text{Tr}[\mathcal{E}_\lambda(\rho_i)\Pi_i]}{\sum_{j=1}^k p_j \text{Tr}[\mathcal{E}_\lambda(\rho_j)\Pi_j]} \geq 1 - \beta_i, \quad i = 1, \dots, k, \quad (5e)$$

where α_i and β_i are user-defined upper bounds on the false positive and false negative error tolerances for identifying ρ_i , and λ denotes the noise level.

CrossQSD provides a tunable compromise between accuracy, confidence, and efficiency. It is conceptually related to maximum-confidence discrimination [13] and worst-case a posteriori probability discrimination [21], but extends these schemes in two crucial ways: first, by explicitly taking noise into account via $\mathcal{E}_\lambda$; and second, by allowing simultaneous, independent control over both false positive and false negative error tolerances through the adjustable parameters α_i and β_i . This flexibility enables a continuous interpolation between the regimes of strict unambiguity and minimal error, offering a unified framework for QSD in the presence of noise.

To evaluate the performance of CrossQSD, we consider three coherent states truncated to an eight-dimensional Hilbert space (corresponding to three qubits) as the predefined noise-free states ρ_i , each assigned an equal prior probability $p_i = 1/3$. A coherent state $|\alpha\rangle$ is defined as the eigenstate of the annihilation operator a , satisfying $a|\alpha\rangle = \alpha|\alpha\rangle$. Specifically, we choose $\alpha = 1, e^{-i\pi/3}$, and $e^{-i2\pi/3}$. The tolerance bounds are set to $\alpha_i = \beta_i = 0.01$.

The noise level λ is varied from 10^{-6} to 10^0 to simulate different degrees of noise contamination. However, when solving the SDP optimization, we assume a *fixed* evaluation noise level of $\lambda_{\text{eval}} = 0.01$ for λ , in order to deliberately examine the effect of a mismatch between the anticipated and actual noise.

The results, shown in Figure 1, are presented in terms of the error-to-success ratio $P_{\text{err}}/P_{\text{succ}} \equiv (1 - P_{\text{succ}})/P_{\text{succ}}$. The error-to-success ratio remains small as long as λ is below λ_{eval} , whereas it increases significantly when λ becomes much larger than λ_{eval} . These findings demonstrate that, as long as the noise level is not

underestimated and the error tolerances α_i and β_i are chosen to be reasonably comparable to λ_{eval} , CrossQSD delivers consistently robust performance, even when the noise level is significantly overestimated due to incomplete knowledge of its characteristics.

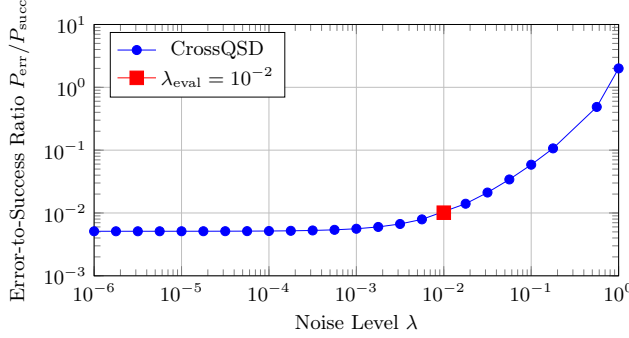


Fig. 1. CrossQSD: Error-to-success ratio $P_{\text{err}}/P_{\text{succ}}$ vs. noise level λ . The predefined states are three coherent states truncated to three qubits; $\alpha_i = \beta_i = 0.01$; $\lambda_{\text{eval}} = 0.01$.

2.3 FitQSD

The other direction for extending UQSD to handle noise is the *FitQSD* approach, which seeks to make the probability distribution of measurement outcomes under noise as close as possible to that of the corresponding noiseless UQSD, according to a chosen notion of “closeness.”

Specifically, consider the joint probability that Alice sends ρ_i while Bob identifies it as ρ_j , given by

$$p(\rho_i, \Pi_j) = p_i \text{Tr}(\rho_i \Pi_j). \quad (6)$$

Let $\{\Pi_i^0\}$ denote the optimal POVM for the noiseless UQSD, yielding the reference probability distribution $\{p(\rho_i, \Pi_j^0)\}$ for all $i, j = 1, \dots, k+1$. In the presence of noise, described by the quantum channel $\mathcal{E}_\lambda$, a QSD strategy with POVM $\{\Pi_i\}$ instead produces $\{p(\mathcal{E}_\lambda(\rho_i), \Pi_j)\}$. The goal of FitQSD is to make these two probability distributions as close as possible, given that the reference $\{p(\rho_i, \Pi_j^0)\}$ is known or solved in advance.

The closeness can be quantified by an L_p -norm, leading to the following convex optimization formulation:

$$\begin{aligned} & \underset{\{\Pi_i\}}{\text{minimize}} && \sum_{i,j=1}^{k+1} \left| p(\rho_i, \Pi_j^0) - p(\mathcal{E}_\lambda(\rho_i), \Pi_j) \right|^\ell, \end{aligned} \quad (7a)$$

$$\text{subject to} \quad \Pi_i \succeq 0, \quad i = 0, \dots, k+1, \quad (7b)$$

$$\sum_{i=0}^{k+1} \Pi_i = I, \quad (7c)$$

where $\ell > 0$ corresponds to the index p of the L_p -norm. Particularly, the cases with $\ell = 1$ and $\ell = 2$ are referred to as *FitQSD-MinL1* (minimizing the L_1 distance) and *FitQSD-MinSS* (minimizing the sum of squares), respectively.

Alternatively, rather than minimizing an L_p -norm directly, one may instead maximize the success probability—similar to CrossQSD—while still maintaining a close match between the noisy and noiseless cases. This leads to the *FitQSD-MECO* (minimizing error with constrained overlap) formulation, expressed as a semidefinite program:

$$\begin{aligned} \underset{\{\Pi_i\}}{\text{maximize}} \quad & P_{\text{succ}} = \sum_{i=1}^k p_i \text{Tr}(\mathcal{E}_\lambda(\rho_i) \Pi_i), \end{aligned} \quad (8a)$$

$$\text{subject to} \quad \Pi_i \succeq 0, \quad i = 1, \dots, k+1, \quad (8b)$$

$$\sum_{i=1}^{k+1} \Pi_i = I, \quad (8c)$$

$$p_i \text{Tr}(\mathcal{E}_\lambda(\rho_i), \Pi_i) \leq p(\rho_i, \Pi_i^0), \quad \text{for } i = 1, \dots, k, \quad (8d)$$

$$p_i \text{Tr}(\mathcal{E}_\lambda(\rho_i), \Pi_j) \geq p(\rho_i, \Pi_j^0), \quad \text{for } i \neq j, \quad i, j = 1, \dots, k. \quad (8e)$$

Here, the reference probabilities $p(\rho_i, \Pi_i^0)$ act as upper bounds that prevent the optimization from overly boosting certain correct-identification rates $p_i \text{Tr}(\mathcal{E}_\lambda(\rho_i) \Pi_i)$ at the expense of others. Similarly, the reference probabilities $p(\rho_i, \Pi_j^0)$ for $i \neq j$ serve as lower bounds that prevent the optimization from excessively suppressing specific misidentification rates $p_i \text{Tr}(\mathcal{E}_\lambda(\rho_i) \Pi_j)$ in favor of others. These comparative bounds constrain the optimized joint probability distribution $\{p(\mathcal{E}_\lambda(\rho_i), \Pi_j)\}$ to remain close to its noiseless reference $\{p(\rho_i, \Pi_j^0)\}$, thereby ensuring consistency and robustness.

To evaluate the performance of FitQSD, we consider the following three two-qubit entangled states with equal prior probabilities:

$$|\psi_i\rangle = \frac{1}{\sqrt{1+a_i^2}} (|b_{i-1}\rangle + a_i |11\rangle), \quad \text{for } i = 1, 2, 3, \quad (9)$$

where $a_1 = 0.2$, $a_2 = 0.5$, $a_3 = 0.7$, and b_i denotes the binary representation of i . The noise level λ is varied from 10^{-6} to 10^{-1} to simulate different noise regimes. We use MOSEK [2] to solve the convex optimization.

The results are presented in Figure 2. To ensure a fair comparison, the closeness between the noisy and noiseless distributions is consistently quantified using the L_2 distance across all three methods. FitQSD-MinL1 and FitQSD-MECO yield nearly identical success probabilities and L_2 distances, whereas FitQSD-MinSS exhibits noticeable deviations in both aspects. All three methods maintain high success probabilities as long as the noise level λ remains relatively low. As λ increases, their performance in P_{succ} gradually deteriorates, with MinSS showing a milder decline. Conversely, the L_2 distance increases for all methods as λ rises, but MinSS performs noticeably worse when λ is small. Therefore, if both success probability and distributional closeness between the noisy and noiseless

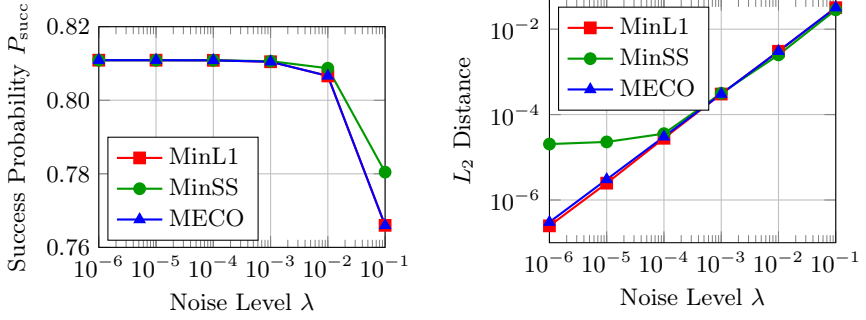


Fig. 2. FitQSD: Success probability P_{succ} (left) and L_2 distance (right) vs. noise level λ . The predefined states are given by Eq. (9). The MOSEK solver precision is set to 10^{-9} .

measurement outcomes are to be optimized, MinL1 and MECO are preferable in the low-noise regime, whereas MinSS performs better under relatively high noise.

It is intriguing—and warrants further investigation—that MinL1 and MECO, though conceptually quite different, yield remarkably similar results, whereas MinL1 and MinSS, which are conceptually very similar, produce noticeably different outcomes. The close agreement between MinL1 and MECO is particularly valuable, as the latter can serve as a practical substitute for the former: MECO is formulated as a semidefinite program and can therefore be solved much more efficiently while achieving nearly the same performance. Furthermore, the FitQSD framework can be naturally extended to other values of ℓ in the L_p -norm, which may provide additional advantages under different noise characteristics—a direction that deserves further exploration.

2.4 Hybrid-objective QSD

In addition to the previously discussed variants, we introduce the *hybrid-objective QSD* scheme, which combines multiple optimization objectives to achieve a balanced trade-off among accuracy, performance, and robustness. This hybrid formulation provides a unified and flexible framework that encompasses and extends conventional approaches such as MED, UQSD, and their noise-aware generalizations. By jointly maximizing the success probability and minimizing the difference between the noisy and noiseless measurement outcomes—with a tunable parameter w adjusting the trade-off between these two competing objectives—the hybrid-objective QSD enables a continuous interpolation between MED⁺ and FitQSD.

The corresponding semidefinite program is formulated, with $w \geq 0$ and $\ell \geq 0$, as

$$\underset{\{H_i\}}{\text{maximize}} \quad \sum_{i=1}^k p_i \text{Tr}(\mathcal{E}_\lambda(\rho_i) H_i) - w \sum_{i,j=1}^{k+1} \left| p(\rho_i, H_j^0) - p(\mathcal{E}_\lambda(\rho_i), H_j) \right|^\ell, \quad (10a)$$

$$\text{subject to } \Pi_i \succeq 0, \quad i = 1, \dots, k+1, \quad (10b)$$

$$\sum_{i=1}^{k+1} \Pi_i = I. \quad (10c)$$

Equivalently, this continuous interpolation can be viewed as one between MED ($w \rightarrow 0$) and UQSD ($w \rightarrow \infty$), since MED^+ has been shown to be equivalent to MED, while FitQSD is designed to reproduce the measurement outcomes of the noiseless UQSD as closely as possible.

To evaluate the performance of the hybrid-objective QSD scheme, we conduct simulations with $\ell = 1$ for two test cases, each involving a different set of predefined states with equal prior probabilities. The first case uses the predefined states given in (9), while the second case involves the two nonorthogonal single-qubit states:

$$|0\rangle \quad \text{and} \quad \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle). \quad (11)$$

The corresponding results are shown in Figures 3 and 4. As before, the L_2 distance is employed to quantify the closeness between the noisy and noiseless probability distributions.

The performance remains robust and consistent over a broad range of relatively low noise levels, where both P_{succ} and the L_2 distance smoothly approach their respective MED values (horizontal dashed lines at the top) as w decreases, and those of the noiseless UQSD (horizontal dotted lines at the bottom) as w increases. As the noise level increases beyond a certain threshold, P_{succ} begins to degrade gradually, while the L_2 distance likewise departs from its low-noise behavior.² Overall, these findings demonstrate that the hybrid-objective QSD scheme offers a reliable and robust continuous interpolation between MED and UQSD under moderate noise conditions.

3 Quantum Circuit Synthesis

Given a POVM solution, we now turn to the problem of implementing a quantum circuit that realizes the corresponding measurement. The synthesis of such a circuit involves two main steps: (i) determining an *isometry*—a norm-preserving linear map—that maps the POVM to a projection-valued measure (PVM) under specified constraints, and (ii) constructing this isometry as an efficient quantum circuit that meets the desired resource and structural requirements.

² As the noise level λ increases, all input states are progressively driven toward the maximally mixed state, i.e., $\mathcal{E}_\lambda(\rho_i) = \rho'_i \xrightarrow{\lambda \rightarrow 1} I/2^n$. In this limit, the noisy states $\{\rho'_i\}$ become nearly identical and therefore essentially indistinguishable. Consequently, the MED (i.e., $w \rightarrow 0$ limit) POVM $\{\Pi_i\}$ reduces to the strategy of always guessing the state with the highest prior probability, and the corresponding success probability P_{succ} approaches $\max_i \{p_i\}$. In contrast, the UQSD (i.e., $w \rightarrow \infty$ limit) POVM $\{\Pi_i\}$ reduces to the strategy of always reporting the inconclusive outcome, and P_{succ} thus approaches zero.

Tailored specifically for QSD, our toolkit identifies the isometry that minimizes the number of required ancilla qubits using a novel approach, while the actual circuit-level realization of the isometry is handled by existing circuit-synthesis and circuit-optimization packages. In the following, we first briefly review conventional circuit synthesis for POVMs and then introduce our new synthesis method designed for QSD, which leverages a modified version of Naimark's dilation theorem, as presented in Theorem 2.

Let $\{\Pi_1, \Pi_2, \dots, \Pi_k\}$ be a POVM on a Hilbert space $\mathcal{H}$, and let $\{P_1, P_2, \dots, P_k\}$ be a rank-1 PVM on an extended Hilbert space $\mathcal{H}' = \mathcal{H} \otimes \mathcal{H}_A$, where $\mathcal{H}_A$ is an ancillary system (dilation space). Naimark's dilation theorem guarantees the existence of an isometry

$$V : \mathcal{H} \rightarrow \mathcal{H}', \quad (12)$$

satisfying

$$V \Pi_i V^\dagger = P_i. \quad (13)$$

By choosing the rank-1 PVM in the form $P_i = I \otimes (|i\rangle_A \langle i|_A)$, the corresponding isometry can then be explicitly constructed as

$$V = \sum_{i=1}^k \sqrt{\Pi_i} \otimes |i\rangle_A, \quad (14)$$

where $\sqrt{\cdot}$ denotes the matrix square root. Once V is known, a quantum circuit can be synthesized using standard isometry circuit synthesis methods, such as the efficient one proposed in [18]. While this approach ensures that all POVM outcomes are encoded in the ancillary system and that the unmeasured subsystem yields the corresponding post-measurement states, it requires an ancillary space of at least dimension k , leading to $\dim \mathcal{H}' \geq k \dim \mathcal{H}$. This is generally wasteful for QSD, where the post-measurement states are largely irrelevant.

Instead of directly applying Naimark's dilation theorem to construct the isometry, we introduce the following modified version, whose proof is provided in [23]:

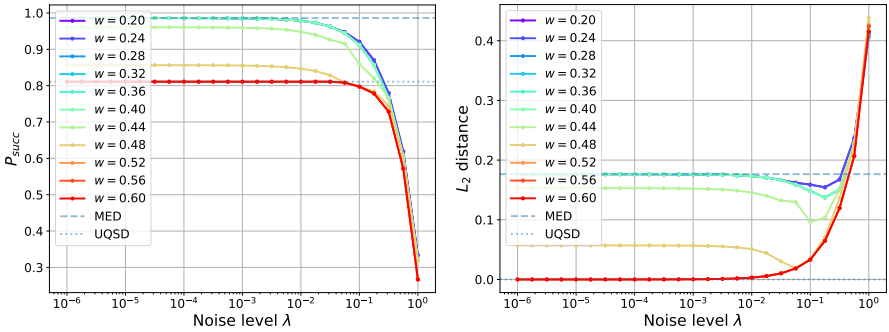


Fig. 3. Hybrid-objective QSD with $\ell = 1$ and the predefined states in Eq. (9): Success probability P_{succ} (left) and L_2 distance (right) vs. noise level λ .

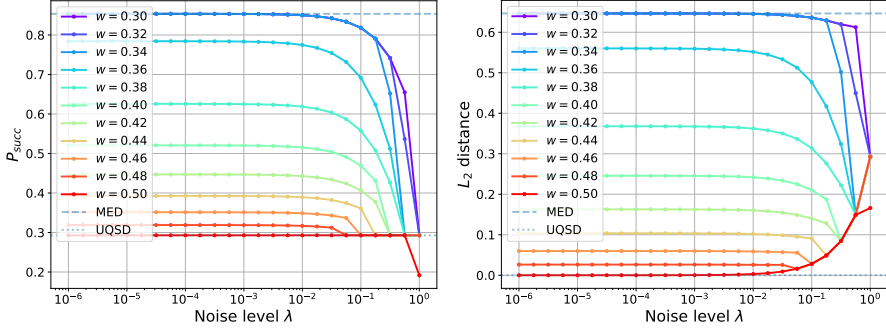


Fig. 4. Hybrid-objective QSD with $\ell = 1$ and the predefined states in Eq. (11): Success probability P_{succ} (left) and L_2 distance (right) vs. noise level λ .

Theorem 2. Let $\Pi = \{\Pi_1, \Pi_2, \dots, \Pi_k\}$ be a POVM on a Hilbert space $\mathcal{H}_A$, and let $\mathbf{P} = \{P_1, P_2, \dots, P_\ell\}$ be a rank-1 PVM on another Hilbert space $\mathcal{H}_{A'}$. If the number of elements in $\mathbf{P}$ satisfies $\ell = \sum_{i=1}^k r_i$, where $r_i = \text{rank}(\Pi_i)$, then there exists an isometry

$$V : \mathcal{H}_A \longrightarrow \mathcal{H}_{A'} \quad (15)$$

such that

$$V \Pi_i V^\dagger = \sum_{j=1}^{r_i} P_{i,j}, \quad \text{for all } i = 1, \dots, k, \quad (16)$$

where the rank-1 projectors in $\mathbf{P}$ are reindexed as

$$\{P_1, P_2, \dots, P_\ell\} = \{P_{1,1}, \dots, P_{1,r_1}, P_{2,1}, \dots, P_{2,r_2}, \dots, P_{k,1}, \dots, P_{k,r_k}\}. \quad (17)$$

This construction allows us to map a POVM to a PVM via the isometry

$$V = \sum_{i=1}^k \sum_{j=1}^{r_i} |g_{i,j}\rangle_{A'} |\tilde{f}_{i,j}\rangle_A. \quad (18)$$

This requires only $\dim \mathcal{H}' \geq \ell$, which is in general substantially smaller than the more wasteful requirement $\dim \mathcal{H}' \geq k \dim \mathcal{H}$ in the previous approach based on Eq. (14), i.e., typically, $\ell \ll k \dim \mathcal{H}$ as $r_i \ll \dim \mathcal{H}$. Moreover, if the states $|g_{i,j}\rangle$ associated with $P_{i,j} = |g_{i,j}\rangle \langle g_{i,j}|$ are chosen to be readily measurable—for instance, $|g_{i,j}\rangle = \left| \sum_{k=1}^{i-1} r_k + j \right\rangle$ as computational-basis states—then the POVM measurement can be effectively implemented as a PVM measurement. Unlike the previous approach, however, each Π_i now might correspond to multiple projectors $P_{i,j}$ for $j = 1, \dots, r_i$, and the post-measurement state is no longer available.

In the context of quantum circuits, by considering $\mathcal{H}_A \subset \mathcal{H}_{A'}$, the POVM can thus be systematically realized through the following procedure: (i) embed any to-be-measured state $\rho \in \mathcal{D}(\mathcal{H}_A)$ into $\mathcal{D}(\mathcal{H}_{A'})$ by appending ancilla qubits; (ii) apply the quantum gate that implements the isometry V ; (iii) perform a

computational-basis measurement; and (iv) whenever the measurement outcome associated with $P_{i,j}$ is obtained, record it as a Π_i event.

We provide a toolkit that automatically identifies the isometry that requires the fewest ancilla qubits and generates the corresponding quantum circuit. In addition, the toolkit supports an approximation mode that achieves significantly higher circuit efficiency at the cost of a small, controllable precision loss. Once a precision threshold³ δ is specified, the terms $|\tilde{f}_{i,j}\rangle$ in Eq. (18) with $\sigma_{i,j} < \delta$ are discarded, yielding a truncated set of $|\tilde{f}_{i,j}\rangle$. Removing these components in Eq. (18) renders V no longer a strict isometry, but standard isometry-synthesis methods can still be applied to obtain an approximate realization based on the truncated set. Although this approximation slightly reduces the overall gate fidelity, it offers a substantial advantage: the number of required ancilla qubits is further reduced, and both the gate count and circuit depth are significantly improved.

Once the isometry V is determined, the toolkit synthesizes the corresponding quantum circuit at the abstract hardware level using the isometry-synthesis method [18] implemented in the `qclib` package [3]. When circuit approximation is desired, the toolkit further applies the approximate quantum compiling technique introduced in [19], which reduces circuit depth while maintaining high fidelity. Finally, if a specific hardware topology is provided through Qiskit’s backend class, the toolkit leverages Qiskit’s `transpile` function to perform circuit resynthesis, generating a hardware-efficient implementation. The overall synthesis workflow is illustrated in Figure 5.

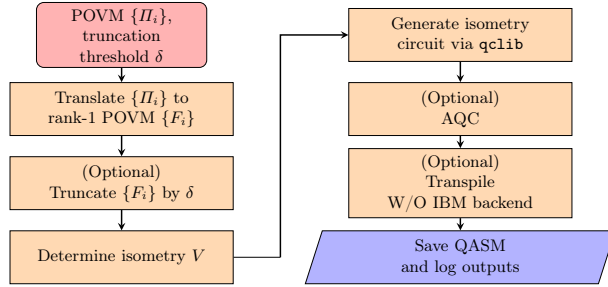


Fig. 5. Quantum circuit synthesis workflow for QSD.

We demonstrate a simple example of circuit synthesis using our toolkit, with the performance results summarized in Table 1. The predefined quantum states are specified in Eq. (9), the prior probabilities are chosen uniformly as $p_1 = p_2 = p_3 = 1/3$, and the noise is turned off ($\lambda = 0$). The isometry corresponding to the optimal MED POVM $\{\Pi_i\}$ is first determined and subsequently

³ The approximation is intended to remove numerically insignificant components of the POVM operators that typically arise from numerical noise in the SDP solver. It is generally difficult to distinguish numerical artifacts from physically meaningful small components in an adaptive manner. For this reason, we expose the truncation threshold as a tunable parameter for the user.

approximated as $\{\tilde{\Pi}_i\}$ by truncating numerically insignificant components of the POVM operators according to a precision threshold $\delta = 10^{-4}$, using the optimization procedures provided by the toolkit. The resulting quantum circuit is then generated using the cosine–sine decomposition implemented in [3] without applying any additional circuit-level optimizations.

As shown in the table, the total POVM rank is significantly reduced by the approximation (i.e., $\sum_i \text{rank}(\tilde{\Pi}_i) < \sum_i \text{rank}(\Pi_i)$). Correspondingly, the circuit complexity decreases substantially, while the QSD performance remains virtually unchanged: the differences in the outcome probabilities $p(\Pi_i|\rho_i)$ are negligible. This highlights a key advantage of the approximation scheme, which yields a considerable reduction in circuit complexity with almost no compromise in discrimination performance.

Table 1. Comparison of circuit complexities, total POVM ranks, and QSD outcome probabilities for the optimal MED, with and without approximation-based optimization (precision threshold $\delta = 10^{-4}$).

Circuit specs	Without Approximation			With Approximation		
Ancilla qubits	2			1		
Single-qubit gates	150			34		
Two-qubit gates	68			15		
Circuit depth	127			25		
POVM total rank	12			6		
Given state ρ_i	Outcome probabilities			Difference in outcome probabilities		
	$p(\Pi_1 \rho_i)$	$p(\Pi_2 \rho_i)$	$p(\Pi_3 \rho_i)$	$p(\tilde{\Pi}_1 \rho_i)$	$p(\tilde{\Pi}_2 \rho_i)$	$p(\tilde{\Pi}_3 \rho_i)$
				$-p(\Pi_1 \rho_i)$	$-p(\Pi_2 \rho_i)$	$-p(\Pi_3 \rho_i)$
$ \psi_1\rangle$ in Eq. (9)	0.99547	1.64E-3	2.89E-3	−1.51E-12	−3.61E-09	−3.78E-09
$ \psi_2\rangle$ in Eq. (9)	1.66E-3	0.98188	1.65E-2	−3.15E-09	−1.36E-11	−1.86E-09
$ \psi_3\rangle$ in Eq. (9)	2.93E-3	1.65E-2	0.98059	−3.03E-09	−1.53E-09	−2.08E-11

4 Toolkit and Demonstration

We present a lightweight Python toolkit that automatically generates quantum circuits for QSD problems according to predefined strategies. The toolkit integrates the SDP solver *CVXPY*, an isometry matrix generator, and the isometry quantum circuit synthesis toolkit *QcLib* within the *Qiskit* framework. Its workflow comprises three stages: (i) construction of a QSD problem instance, specifying the quantum states with their prior probabilities and the noise model; (ii) synthesis of POVM operators according to predefined QSD strategies, which solves the POVM operators using *CVXPY* and optionally provides approximate results; and (iii) quantum-circuit synthesis, which outputs the resulting `QuantumCircuit` object in *Qiskit* through *QcLib*.

In the following, we demonstrate a typical usage of our toolkit. The benchmarks for the computational efficiency of the functions in the toolkit are presented in [23].

4.1 Problem instance construction

This step defines the basic configuration of the QSD problem using the `ProblemSpec` class. An example usage is shown below:

```
from flow.solve_mix import *

# initialize the QSD problem instance
qsd_problem = ProblemSpec(num_qubits, num_states)

# set the prior probabilities by a numpy array
qsd_problem.prior_prob = np.ones(num_states) * (1 / num_states)

# specify the quantum states
qsd_problem.set_states(state_type, states)
```

As shown in the example, the class initialization is intentionally designed to allow users to specify the number of quantum states to be discriminated and the number of qubits for each state before defining the actual quantum states. When users need to replace the current set of quantum states, the `set_states` function can be used, eliminating the need to reinitialize a new `ProblemSpec` object. Additional information, such as prior probabilities, can also be specified later upon needed.

4.2 POVM synthesis with an optimization strategy

This step selects an optimization strategy and uses it to synthesize the POVM for the previously defined problem. An example usage is shown below (continuing from the previous code block).

```
from flow.build_circuits import *

crossqsd_prob = apply_crossQSD(
    qsd_problem,
    cvxpy_settings={"solver": cp.SCS, "verbose": False},
    alpha=[0.02] * qsd_problem.num_states,
    beta=[0.02] * qsd_problem.num_states
)

povm_vectors = get_povm_vectors(crossqsd_prob)
povm_ckt = POVMCircuit(povm_vectors=np.stack(povm_vectors))
povm_ckt.fix()
povm_ckt.build_circuit()
```

We provide a variety of commonly used optimization strategies in the `flow.solve_mix` package, which are summarized in Table 2. The optimization process is implemented using CVXPY to obtain the optimal POVM corresponding to the selected strategy. When using an optimization function, users must specify the target `ProblemSpec` instance, the configuration of the CVXPY solver,

and any additional constraints required by the chosen optimization strategy, as described in Section 2 and listed in Table 2.

Table 2. Python function interfaces for the QSD formulations in the `flow.solve_mix` package

function name	state type	QSD	extra arguments
<code>apply_Eldar</code>	state vector	Optimal UQSD	<code>beta</code>
<code>med_problem</code>	density matrix	MED	(None)
<code>med_plus_problem</code>	density matrix	MED ⁺	(None)
<code>apply_frio</code>	density matrix	FRIO	<code>p_inc_lb</code>
<code>apply_crossQSD</code>	density matrix	CrossQSD	<code>alpha, beta</code>
<code>min_l1_problem</code>	density matrix	FitQSD-minL1	<code>ideal_distrib</code>
<code>min_ss_problem</code>	density matrix	FitQSD-minSS	<code>ideal_distrib</code>
<code>meco_problem</code>	density matrix	FitQSD-MECO	<code>ideal_distrib</code>
<code>hybrid_obj_problem</code>	density matrix	Hybrid-objective	<code>ideal_distrib, param_a</code>

After executing a selected optimizer (e.g., the `apply_crossQSD` function in the example), a `POVMCircuit` instance can be created from the POVM vectors produced by the optimizer. To enable the use of fewer qubits in circuit construction, users can explicitly specify the `num_qubits` attribute. Finally, the `fix()` function completes the POVM by adding residual components above a small singular-value threshold.

4.3 Quantum circuit synthesis and resynthesis

If the user requires a gate-level implementation, the compilation core can be invoked to generate the corresponding quantum circuit. An example usage is shown below (continuing from the previous code block).

```
from flow.resynth import resynth_aqc
qc = povm_ckt.build_circuit("ccd")
qc = resynth_aqc(qc)
```

The `build_circuit` function supports three synthesis schemes, `ccd`, `csd`, and `knill`, and internally employs the `qclib` package to generate the corresponding quantum circuits. Note that when using the `ccd` scheme, the circuit is first extended to a full unitary before decomposition, spending additional computing time.

Once the quantum circuit is generated, the estimated number of CNOT gates is displayed to the console for immediate review. The synthesized circuit is returned as a Qiskit `QuantumCircuit` instance for further use, and the provided `resynth_aqc()` function enables approximate circuit resynthesis via Qiskit’s `transpile()` interface.

5 Case Study: UQSD for Coherent States in Noisy Quantum Circuit Simulators

In this section, we demonstrate the use of our toolkit on quantum computers and simulate the QSD experiment using Qiskit's noisy quantum circuit simulator. For a fair performance assessment, we adopt the UQSD scheme with optimal POVM operators as the discrimination strategy.

In the experiments, we consider three coherent states defined in Section 2.2, with parameters $\alpha = 1, e^{i\frac{2\pi}{3}}, e^{i\frac{4\pi}{3}}$. Since coherent states are intrinsically infinite-dimensional, they must be truncated to a finite Hilbert space for quantum circuit implementation. In the following processes, those coherent states are truncated into subspaces ranging from two to six qubits. The corresponding phase-space representations of the truncated states are shown in Figure 6, which illustrates how the overlap among the three states diminishes as the truncation dimension increases.

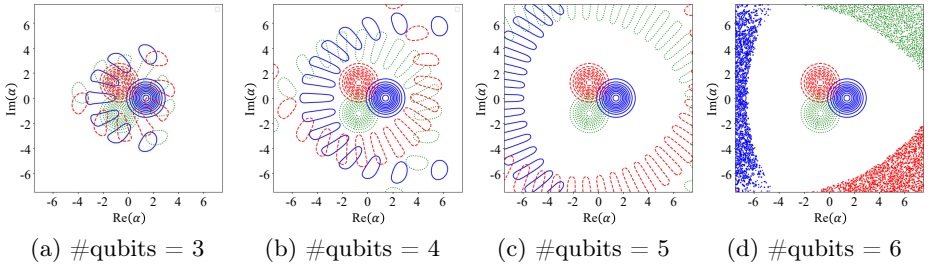


Fig. 6. Wigner functions of truncated coherent states represented with 3 to 6 qubits. The blue, green, and red dots correspond to the states $|\alpha\rangle$ with $\alpha = 1, e^{i\frac{2\pi}{3}},$ and $e^{i\frac{4\pi}{3}},$ respectively. All figures are generated using QuTiP [22].

For the QSD implementation, the optimal POVM operators are first obtained using `apply_Eldar`, after which the corresponding quantum circuits are synthesized with `build_circuit("ccd")`. The resulting circuits are then resynthesized into approximate forms using `approx_circuit`, which adopts the AQC optimization settings from Qiskit's `transpile()` function. The average circuit depth, two-qubit gate count, and state fidelity before and after approximation are summarized in Table 3 for the three representative states. Notably, no ancillary qubits are required in the gate-level synthesis, as the total ranks of the POVM operators in this example are equal to three.

Finally, we perform the QSD protocol for the six-qubit truncated states using Qiskit's noisy quantum circuit simulator, `Qiskit_Aer`. The simulator is configured to emulate depolarization noise on every two-qubit gate, defined as

$$\mathcal{E}(\rho^{(2)}) = (1 - \lambda)\rho^{(2)} + \lambda \frac{I^{(2)}}{2^n}, \quad (19)$$

where λ denotes the level of depolarizing noise, n is the total number of qubits in the circuit, and $\rho^{(2)}$ is the two-qubit reduced density matrix that after gate appli-

Table 3. Comparison of quantum circuit synthesis methods for discriminating three coherent states. Here #2Q means the number of two-qubit gates, which is also the number of CNOT gates in this case. The numbers for the state fidelity and process fidelity are rounded down to the fifth digit.

#Qubit	State Fidelity	Original		Resynthesis			Approx_deg = 1.00			Approx_deg = 0.97			AQC (#2Q = 10)		
		Depth	#2Q	Depth	#2Q	Process Fidelity	Depth	#2Q	Process Fidelity	Depth	#2Q	Process Fidelity	Depth	#2Q	Process Fidelity
2	0.97803	84	41	45	20	1.00000	29	14	0.99999	29	14	0.59409	21	10	0.35564
3	0.99998	430	218	225	100	0.99999	83	61	0.99992	83	61	0.88925	15	10	0.96252
4	0.99999	2034	1025	1009	444	0.99999	253	252	0.99989	253	252	0.13754	11	10	0.32505
5	0.99999	8916	4474	4273	1868	0.99999	817	1020	0.99990	817	1020	0.00032	9	10	0.25978
6	0.99999	37220	18634	17585	7660	0.99999	2729	4091	0.99984	-	-	-	-	-	-

cation. Simulations are performed for various λ values ranging from 10^{-6} to 10^{-1} , with each configuration executed using 1024 shots. The measurement outcomes are subsequently post-processed to estimate the empirical success probabilities.

The experimental results are presented in Figure 7. The success probabilities for discriminating each truncated six-qubit coherent state remain close to the noiseless values for $\lambda < 10^{-3}$, but drop sharply when $\lambda > 10^{-3}$. These findings confirm our earlier observation that UQSD fails immediately once the noise exceeds a critical threshold.

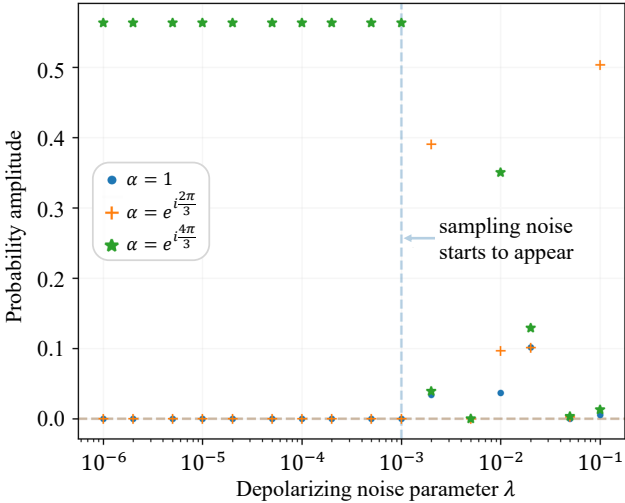


Fig. 7. Experimental success probabilities for discriminating each six-qubit-truncated coherent states vs. noise level λ .

Acknowledgments

This work was supported in part by the National Science and Technology Council of Taiwan under grants 114-2119-M-002-020, the NTU Center of Data Intelligence: Technologies, Applications, and Systems under grant NTU-114L900903, and the NTU Core Consortium Project NTU-CC114L895002.

Data Availability Statement

The source code, benchmark data, and proofs of the theorems used in this study are publicly available at <https://zenodo.org/records/17493285>.

References

1. Agrawal, A., Amos, B., Barratt, S., Boyd, S., Diamond, S., Kolter, J.Z.: Differentiable convex optimization layers. *Advances in neural information processing systems* **32** (2019)
2. ApS, M.: MOSEK Optimizer API for Python 11.0.29 (2025), <https://docs.mosek.com/11.0/pythonapi/index.html>
3. Araujo, I.F., Araújo, I.C.S., da Silva, L.D., Blank, C., da Silva, A.J.: Quantum computing library (February 2023), <https://github.com/qclib/qclib>
4. Arecchi, F.T., Courtens, E., Gilmore, R., Thomas, H.: Atomic coherent states in quantum optics. *Physical Review A* **6**(6), 2211 (1972)
5. Bae, J.: Structure of minimum-error quantum state discrimination. *New Journal of Physics* **15**(7), 073037 (2013)
6. Bae, J., Kwek, L.C.: Quantum state discrimination and its applications. *Journal of Physics A: Mathematical and Theoretical* **48**(8), 083001 (2015)
7. Bagan, E., Muñoz-Tapia, R., Olivares-Rentería, G., Bergou, J.: Optimal discrimination of quantum states with a fixed rate of inconclusive outcomes. *Physical Review A—Atomic, Molecular, and Optical Physics* **86**(4), 040303 (2012)
8. Barnett, S.M., Croke, S.: Quantum state discrimination. *Advances in Optics and Photonics* **1**(2), 238–278 (2009)
9. Bennett, C.H., Brassard, G.: Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science* **560**, 7–11 (2014). <https://doi.org/10.1016/j.tcs.2014.05.025>
10. Bergou, J.A.: Discrimination of quantum states. *Journal of Modern Optics* **57**(3), 160–180 (2010)
11. Boyd, S.P., Vandenberghe, L.: *Convex optimization*. Cambridge university press (2004)
12. Chefles, A.: Quantum state discrimination. *Contemporary Physics* **41**(6), 401–424 (2000)
13. Croke, S., Andersson, E., Barnett, S.M., Gilson, C.R., Jeffers, J.: Maximum confidence quantum measurements. *Physical Review Letters* **96**(7) (2006). <https://doi.org/ARTN07040110.1103/PhysRevLett.96.070401>
14. Eldar, Y.C.: A semidefinite programming approach to optimal unambiguous discrimination of quantum states. *IEEE Transactions on Information Theory* **49**(2), 446–456 (2003). <https://doi.org/10.1109/TIT.2002.807291>
15. Gazeau, J.P.: *Coherent states in quantum physics*. Wiley Online Library (2009)
16. Grant, M., Boyd, S., Ye, Y.: Disciplined convex programming. In: *Global optimization: From theory to implementation*, pp. 155–210. Springer (2006)
17. Herzog, U.: Optimal state discrimination with a fixed rate of inconclusive results: Analytical solutions and relation to state discrimination with a fixed error rate. *Physical Review A—Atomic, Molecular, and Optical Physics* **86**(3), 032314 (2012)
18. Iten, R., Colbeck, R., Kukuljan, I., Home, J., Christandl, M.: Quantum circuits for isometries. *Physical Review A* **93**(3), 032318 (2016). <https://doi.org/10.1103/PhysRevA.93.032318>

19. Ivashkov, P., Uchegara, G., Jiang, L., Wang, D.S., Seif, A.: High-fidelity, multiqubit generalized measurements with dynamic circuits. *PRX Quantum* **5**(3), 030315 (2024). <https://doi.org/10.1103/PRXQuantum.5.030315>
20. Ko, H., Choi, B.S., Choe, J.S., Youn, C.J.: Advanced unambiguous state discrimination attack and countermeasure strategy in a practical B92 QKD system. *Quantum Information Processing* **17**(1) (2018). <https://doi.org/ARTN1710.1007/s11128-017-1784-7>
21. Kosut, R.L., Walmsley, I., Eldar, Y., Rabitz, H.: Quantum state detector design: Optimal worst-case a posteriori performance (2004), <https://arxiv.org/abs/quant-ph/0403150>
22. Lambert, N., Giguère, E., Menczel, P., Li, B., Hopf, P., Suárez, G., Gali, M., Lishman, J., Gadhvi, R., Agarwal, R., Galicia, A., Shammah, N., Nation, P., Johansson, J.R., Ahmed, S., Cross, S., Pitchford, A., Nori, F.: QuTiP 5: The quantum toolbox in Python (2024), <https://arxiv.org/abs/2412.04705>
23. Ma, C.K., Chen, B.H., Chen, T.F., Chiou, D.W., Jiang, J.H.R.: Supplemental Material for “Error-Tolerant Quantum State Discrimination: Optimization and Quantum Circuit Synthesis” (2026), <https://zenodo.org/records/17493285>
24. Namark, M.: On the representations of additive operator set functions. In: *CR (Doklady) Acad. Sci. USSR*. vol. 41, pp. 359–361 (1943)
25. Qiu, D.: Minimum-error discrimination between mixed quantum states. *Physical Review A—Atomic, Molecular, and Optical Physics* **77**(1), 012328 (2008)
26. Scarani, V., Acín, A., Ribordy, G., Gisin, N.: Quantum cryptography protocols robust against photon number splitting attacks for weak laser pulse implementations. *Physical Review Letters* **92**(5) (2004). <https://doi.org/ARTN05790110.1103/PhysRevLett.92.057901>
27. Vandenberghe, L., Boyd, S.: Semidefinite programming. *SIAM review* **38**(1), 49–95 (1996)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



SV-COMP



Evaluating Software Verifiers for C, Java, and SV-LIB (Report on SV-COMP 2026)

Dirk Beyer¹✉ and Jan Strejček²

¹ LMU Munich, Munich, Germany

² Masaryk University, Brno, Czechia

Abstract. The *Competition on Software Verification (SV-COMP)* regularly evaluates software verifiers and witness validators processing programs in C and Java. SV-COMP 2026 evaluated 61 verifiers and 16 validators for C programs and 11 verifiers and 3 validators for Java programs. Out of these, 43 verifiers and 13 validators participated with an active support of teams led by 44 different representatives from 12 countries. The verification track of the competition was executed on a benchmark set of 36 402 verification tasks with C programs and 6 different specifications and 1 731 verification tasks with Java programs and 2 different specifications. The validation track analyzed 229 118 witnesses generated in the verification track for C programs and also 135 handcrafted witnesses. On top of that, SV-COMP 2026 considered also 254 verification tasks in the recently introduced format SV-LIB and evaluated 3 verifiers and 1 validator for this format. Moreover, there was also a demo category sponsored by Huawei, consisting of selected verification tasks with concurrent C programs. To keep our growing competition sustainable and up to date, we made several changes in its processes and settings.

Keywords: Formal Verification · Program Analysis · Competition · Software Verification · Verification Tasks · Verification Witnesses · Witness Validation · Benchmark · Specification · C Language · Java Language · SV-LIB · SV-COMP · SV-Benchmarks · BENCHEXEC · CoVeriTeam

1 Introduction

This report presents the objectives, processes, rules, participants, and results of the 15th *Competition on Software Verification* (<https://sv-comp.sosy-lab.org/2026>) (SV-COMP 2026), which was again the largest comparative evaluation ever in this area. This paper extends the series of competition reports (see footnote)

This report extends the series [20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 50] of previous reports on SV-COMP.

Reproduction packages are available on Zenodo (see Table 4).

✉ dirk.beyer@sosy-lab.org

documenting the growth and evolution of the competition over these 15 years. This year, we focus on the changes in the competition done in 2026, including the newly added categories and modifications in witness support.

The objectives of the competitions were discussed earlier (1–4 [26]) and extended over the years (5–6 [27] and 7 [31]):

1. provide an overview of the state of the art in software-verification technology and increase visibility of the most recent software verifiers,
2. establish a repository of software-verification tasks that is publicly available for free use as a standard benchmark suite for evaluating verification software,
3. establish standards that make it possible to compare different verification tools, including a property language and formats for the results,
4. accelerate the transfer of new verification technology to industrial practice by identifying the strengths of the various verifiers on a diverse set of tasks,
5. educate PhD students and others on performing reproducible benchmarking, packaging tools, and running robust and accurate research experiments,
6. provide research teams that do not have sufficient computing resources with the opportunity to obtain experimental results on large benchmark sets, and
7. conserve tools for formal methods for later reuse by using a standardized format to announce archives (via DOIs), default options, contacts, competition participation, and other meta data in a central repository.

The SV-COMP 2020 report [27] discusses the achievements of the SV-COMP competition so far with respect to these objectives.

Related Competitions. SV-COMP is one of many competitions that measure progress of research in the area of formal methods [18]. Competitions can lead to fair and accurate comparative evaluations because of the involvement of the developing teams. The competitions most related to SV-COMP are RERS [98], VerifyThis [80], Test-Comp [32], and TermCOMP [90]. The SV-COMP 2020 report [27] provides a more detailed discussion.

Quick Summary of Changes. We aim to keep the setup of the competition stable. On the other hand, the competition setup has to reflect the advances of the research area. The edition 2026 brought the following changes:

- The tool registration process and qualification criteria were simplified.
- Until 2025, the competition jury consists of representatives of the tools qualified for the competition. Since 2026, the jury contains representatives of the tools registered to the competition and their duties are slightly reduced.
- We introduced and applied systematic naming of categories. Each category now starts with the programming language of the contained programs (C, Java, or SV-LIB). The names of base categories then continue with the specification property of the contained verification tasks. We also renamed some properties to keep the names coherent. In particular, we renamed the properties `assert_java` and `runtime-exception` considered in Java benchmarks to `valid-assert` and `no-runtime-exception`, respectively.

- The meta category *FalsificationOverall* was renamed to *C.FalseOverall* and now it contains also benchmarks with the **termination** property. We introduced the dual meta category called *C.TrueOverall*.
- We had a demo category *C.Huawei-Concurrency-Challenges* sponsored by Huawei, consisting of selected tasks from the category *C.Concurrency*.
- The number of considered verification tasks again increased in both C and Java languages: the number of C tasks increased from 33 353 in 2025 to 36 402 and the number of Java tasks increased from 1 345 in 2025 (including tasks in demo categories) to 1 731.
- Besides benchmarks with programs in C and Java, we added a demo category with 254 verification tasks in the recently introduced format SV-LIB [40].
- We updated the support of witness formats. In particular, we now support the [witness format 2.1](#) and we discontinued support for correctness witnesses in format 1.0 (with the exception of witnesses for Java programs and the witnesses produced by verifiers without active team support).
- We added a new category *C.termination.ViolationWitnesses* of 17 handcrafted validation tasks with witnesses of program non-termination in format 2.1.
- The CPU time limit for validation of a correctness witness was lowered from 15 min to 5 min.
- We rewrote the scripts that post-process the output of all evaluated tools to make them more modular, efficient, and maintainable. The intermediate data are now stored in CSV instead of XML files.
- As ETAPS 2025 was held in Canada and many community members were not able to attend the traditional SV-COMP session there, we organized a one-day workshop on the current state and future development of SV-COMP and Test-Comp in April 2025 at Frauenchiemsee. Following its success, the second [SV-COMP/Test-Comp Workshop](#) was held on March 17, 2026 in Munich.

2 Organization and Processes

Organization. For the first 13 years, SV-COMP was ran by a single organizer (Dirk Beyer 2012–2017, Tomáš Vojnar 2018, Dirk Beyer 2019–2024). To handle its growing complexity, an organization committee was established before SV-COMP 2025. All members of the committee were also working on SV-COMP 2026. More precisely, the competition had two chairs (Dirk Beyer and Jan Strejček) and committee members in charge of benchmark quality (Zsófia Ádám, Raphaël Monat, Simmo Saan, and Frank Schüssele), category structure (Thomas Lemberger), infrastructure development (Philipp Wendler, Po-Chun Chien, Marek Jankola, Henrik Wachowitz, Matthias Kettl, and Marian Lingsch-Rosenfeld), and reproducibility (Levente Bajczi). For 2026, the committee was extended by members responsible for tool qualification (Paulína Ayaziová, Matthias Heizmann, Marian Lingsch-Rosenfeld, Felix Mächtle, Raphaël Monat, Malte Mues, and Jan-Niclas Serr).

Procedure. SV-COMP is an open competition (also known as comparative evaluation). Verification tasks and handcrafted validation tasks are publicly available in a repository ([Table 5](#)) where anyone can contribute. The competition has basically

two phases: *training* and *evaluation*. During the training phase, participating teams can repeatedly submit new versions of their tool. The organizers run the tool on relevant tasks and provide the results to the whole community. The participants can inspect the results, fix bugs in their tools and submit a new version, or report an issue with some tasks. The set of verification tasks and handcrafted validation tasks is *frozen* approximately two weeks before the the end of the training phase. In the evaluation phase, the tools are again executed on all relevant tasks and the participants are asked for an inspection of the results. They can challenge the validity of some tasks and suggest modifications of these tasks. Finally, the tasks modified after the freezing are excluded from the score computation (they are marked as *void*) and the results are announced on the competition web site.

Competition Jury. The competition jury reviews the competition contribution papers and helps the organizer with resolving any disputes that might occur. The jury consists of the SV-COMP chairs and one representative of each *registered* verifier and validator (a tool registered both as a verifier and a validator can have two representatives). In the previous editions, the jury consisted of the representatives of *qualified* tools and was responsible also for checking whether the tools registered to the next edition of SV-COMP fulfill the qualification criteria. Before this edition, we transferred this duty to the organization committee members mentioned above. The current representatives are listed in [Tables 6 and 7](#). The jury includes one additional member, A. R. Koçal (Technische Universität München, Germany), representing the tool [GOBLINT-PAR](#) which was registered, but not qualified to the competition. The current jury is also listed on the web site (<https://sv-comp.sosy-lab.org/2026/committee.php>).

3 Tasks, Workflow, and Scoring

Verification and Validation Tasks. A *verification task* consists of a program, a property to be verified, and the expected verification result. Tools for solving verification tasks are called *verifiers*. A *validation task* is a program, a property, and a witness of the program correctness or property violation to be validated. A typical validation task is a verification task extended with a witness generated by a verifier that solved the verification task. Some validation tasks (e.g., the handcrafted ones) contain also the expected validation result. Tools for solving validation tasks are called *validators*. SV-COMP 2026 used the [task-definition format in version 2.1](#) to denote the verification and validation tasks.

Programs and Properties. In connection with C programs, we considered 6 different properties: *unreachability of a given function* (referenced as `unreach-call` in the competition), *memory safety* (`valid-memsafety`) composed of three sub-properties saying that all pointer dereferences are valid (`valid-deref`), all memory deallocations are valid (`valid-free`), and all allocated memory is tracked (`valid-memtrack`), *memory cleanup* (`valid-memcleanup`) saying that all allocated memory is deallocated before the program terminates, *no overflow* (`no-overflow`) saying that operations on signed integers never overflow, *no data*

race (**no-data-race**) saying that the program does not contain any data race, and *termination* (**termination**) saying that the program always terminates. Note that a program is considered memory safe only if it satisfies all three subproperties. The subproperties are used in particular to report what is violated if a program is not memory safe. For Java programs, we considered the properties *assertion validity* (**valid-assert**) and *no runtime exception* (**no-runtime-exception**). In the previous years, these properties were referenced as **assert_java** and **runtime-exception**, respectively. We refer to the conference web page for precise definition of these properties (<https://sv-comp.sosy-lab.org/2026/rules.php>).

Additionally, SV-COMP 2026 also considered verification tasks in the recently introduced format for software-verification called SV-LIB [40]. One of the key features of the format is clear formal semantics, painfully missing for languages like C and Java. The format allows us to describe not only imperative programs, but also specifications and witnesses. The program specification is given by *tag annotations* which can describe a large class of safety and liveness properties. To check that an SV-LIB program satisfies its specification actually means to check that the tag annotations are correct. In SV-COMP, we call this property **correct-tags**.

Witnesses. SV-COMP 2026 supported several different formats for verification witnesses, depending on the programming language, considered property, specific program features, and verification result. In general, for correctness witnesses of C programs, we used witness formats 2.0 [8] and 2.1. The main benefit of the format 2.1 is added support of concurrent programs [83] and **termination** property [43, 154]. The format 2.1 allows correctness witnesses to contain *function contracts* and claim *inductivity* of invariants [95], but SV-COMP 2026 did not support these features as they were introduced only very shortly before the competition and did not have sufficient tool support. Note that after SV-COMP 2025, the competition abandoned the format 1.0 [37] for correctness witnesses of C programs with one exception: *inactive* verifiers (i.e., verifiers without active team support) can produce correctness witnesses in format 1.0 as in the previous year. For violation witnesses of C programs, we used witness formats 1.0, 2.0, and 2.1. For witnesses of Java programs, we used witness format 1.0. For SV-LIB tasks, we used witnesses in the SV-LIB 1.0 format. More information about supported witness formats for various classes of verification tasks is provided later.

Categories. The verification tasks are divided into *base categories* reflecting the programming language, the considered property, and loosely also some specific program features or the source of the benchmarks. Before SV-COMP 2026, we renamed all base categories such that now their names start with the prefix of the form *<language>.<property>*. Base categories are accumulated into *meta categories* whose names now also start with the prefix *<language>*.

For the C language, there are two levels of meta categories. The top level contains the categories *C.Overall*, *C.FalseOverall*, and *C.TrueOverall* of all C verification tasks. The category *C.FalseOverall* was previously called *FalsificationOverall* and contained only C tasks with safety properties (i.e., no **termination** property). The category shows bug-finding ability of participating tools by using a specific scoring schema explained below. The category *C.TrueOverall* was con-

sidered for the first time and it is dual to *C.FalseOverall*. Intuitively, its scoring schema shows ability to prove program correctness. One of the long-term goals of the SV-COMP community is to extend the benchmark set with more industrial verification task. As a step in this direction, the meta category *C.SoftwareSystems* was extended with one new base category *C.termination.SoftwareSystems-uthash*. Further, we improved and added new benchmarks to the category *C.unreach-call.SoftwareSystems-Intel-TDX-Module* introduced in the previous edition, such that now it contains 836 tasks compared to 146 in SV-COMP 2025. More details about this effort can be found in a case-study article [34], published in this volume as well. This improvement included introduction of a new competition-specific function `__VERIFIER_nondet_memory(void *, size_t)` which havoc a given memory block (the first argument must be a valid pointer to the start of a memory block and the second argument must give the size). To increase the visibility and encourage support for this new function, more tasks to check the handling of this function will be introduced in the next edition. Finally, the meta category *C.ValidationCrafted* was extended with new base category *C.termination.ViolationWitnesses* of 17 crafted validation tasks with witnesses of **termination** violation in format 2.1.

For the Java language, the category with the **no-runtime-exception** property switched from demo category to regular base category. The overall number of Java verification tasks significantly increased from 1345 tasks in SV-COMP 2025 to 1731 tasks in SV-COMP 2026. One of the sources for extension was the tool ARG-V, which can pull Java software projects from GitHub repositories and readily prepare them as benchmark set. The tool is also described in this volume [122]. The structure of all regular base and meta categories is shown in Fig. 1.

SV-COMP 2026 considered also some demo categories shown in Fig. 2. Four base demo categories accumulated in meta category *C.Huawei-Concurrency-Challenges* contain selected verification tasks from *C.Concurrency*. The meta category is sponsored by Huawei, who provided prize money for the top three verifiers in this category (excluding tools with at least one Huawei employee in the team, meta verifiers, and verifiers without active team support) in the amount of 3 000 EUR for the winner, 1 500 EUR for the second, and 500 EUR for the third place. The remaining three base demo categories accumulated in *SV-LIB.Overall* consist of SV-LIB tasks. They were treated as demo categories as they were announced very shortly before the competition deadlines.

The number of verification tasks in individual meta categories is shown in Tables 10, 11, and 12 presenting the results of the verification track. Finally, the competition web site (<https://sv-comp.sosy-lab.org/2026/benchmarks.php>) provides a detailed description of the categories.

Besides the verification track, the competition has also a validation track for validators of verification witnesses for C programs. We used the validation tasks with witnesses generated by verifiers on the verification tasks from *C.Overall* and the handcrafted validation tasks in *C.ValidationCrafted*. All tasks in the validation track were divided into 3 subtracks based on the witness type and used format, namely (a) correctness witnesses in formats 2.0 and 2.1, (b) violation

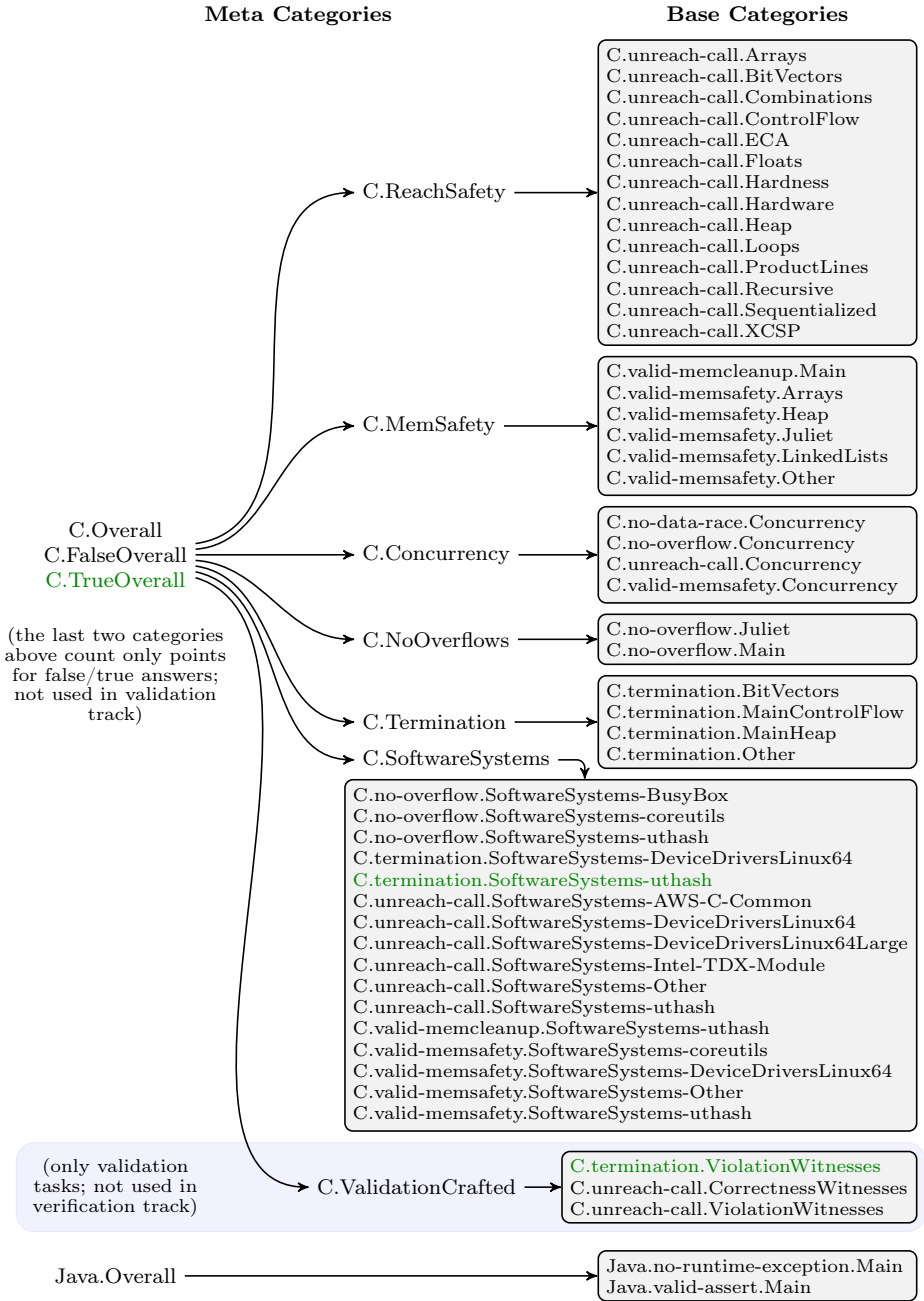


Fig. 1: Regular categories of SV-COMP 2026; new categories are green

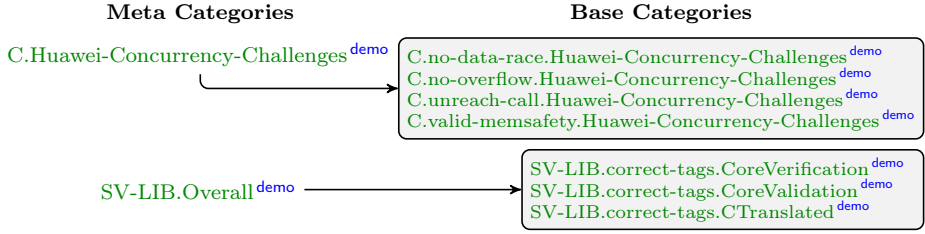


Fig. 2: Demo categories of SV-COMP 2026; all of them are new

Table 1: Supported witness formats in individual base categories; ‘–’ indicates that witnesses are not supported, ^{demo} means that the witnesses are supported only in *demo mode*, (1.0) denotes *legacy support* relevant only for inactive verifiers

Base Category		Correctness	Violation
Prefix	Suffix	Witnesses	Witnesses
<i>C.unreach-call.</i>	<i>Arrays, Floats, Heap</i>	–	1.0, 2.0, 2.1
	<i>Concurrency, Huawei*</i>	2.1 ^{demo}	1.0
	all others	(1.0), 2.0, 2.1	1.0, 2.0, 2.1
<i>C.valid-memsafety.</i>	<i>Concurrency, Huawei*</i>	–	1.0
	all others	–	1.0, 2.0*, 2.1*
<i>C.valid-memcleanup.</i>	all	–	1.0
<i>C.no-overflow.</i>	<i>Concurrency, Huawei*</i>	2.1 ^{demo}	1.0
	all others	(1.0), 2.0, 2.1	1.0, 2.0, 2.1
<i>C.no-data-race.</i>	all	–	1.0
<i>C.termination.</i>	all	2.1 ^{demo}	1.0, 2.1
<i>Java.valid-assert.</i>	<i>Main</i>	1.0 ^{demo}	1.0
<i>Java.no-runtime-exception.</i>	<i>Main</i>	–	1.0
<i>SV-LIB.correct-tags.</i>	all	SV-LIB	SV-LIB

* The formats 2.0 and 2.1 support only violation of the subproperties **valid-deref** and **valid-free** of **valid-memsafety**. If the violated subproperty is **valid-memtrack**, the format 1.0 has to be used.

witnesses in formats 2.0 and 2.1, and (c) violation witnesses in format 1.0. In each of these subtracks, the validation tasks were organized in the category structure corresponding to *C.Overall* and its subcategories, where each validation task generated by a verifier is in the base category determined by the corresponding verification task. Due to limitations of the witness formats and current validators, SV-COMP 2026 supported witnesses in individual formats only for selected properties and categories, as shown in Table 1. Consequently, some categories in some subtracks did not contain any validation task. The empty categories were removed from the category structure of each subtrack.

Table 1 clearly identifies several important research goals for the verification community: Develop and extend witness formats and validation tools for categories with insufficient validation support.

Competition Workflow. Roughly speaking, the inputs of the competition are *verification tasks* and, for each verifier and witness validator, the participating team creates (a) a *tool archive* on Zenodo from where the tool is downloaded, (b) a file `<tool>.yaml` at the *Formal-Methods Tools* repository with basic information about the tool, the name of its jury representative, identifier of the Zenodo archive, etc., (c) a *tool-info module* in the **BENCHEXEC** repository that specifies the interface for running the tool and interpreting its result, and (d) a merge request to the file `category-structure.yaml` in the *Benchmark Definitions* repository that declares the categories the tool participates in. The links to the mentioned repositories are in [Table 5](#).

The verification track proceeds as follows. The competition scripts run each verifier on all relevant verification tasks given by `category-structure.yaml`. If the verifier solves a task, it returns either `TRUE` meaning that the program satisfies the given property or `FALSE` meaning that the program violates the property. The output `TRUE` is accompanied with a *correctness witness* and the output `FALSE` with a *violation witnesses* in some of the formats supported by SV-COMP for the corresponding category (see [Table 1](#)). The table reflects the fact that the current witness infrastructure is not sufficiently developed to handle correctness witnesses of some verification tasks (e.g., tasks with C programs containing arrays, floats, heap manipulation, or parallelism, and tasks with properties `valid-memsafety` or `valid-memcleanup`). Correctness witnesses in some categories are thus not supported at all (denoted by ‘-’ in the table). Nevertheless, the situation is slowly improving. For example, the recently introduced witness format 2.1 supports parallel programs and **termination**. Because only very few validators can currently handle this format, correctness witnesses in some categories are supported only in *demo mode*. This means that witnesses can be generated and they are analyzed by available validators, but the results of the validation do not play any role in the score computation. The demo mode is also used for correctness witnesses of *Java.valid-assert.Main* due to a limited number of suitable validators. Further, there are some inactive verifiers of C programs that do not produce required correctness witnesses in format 2.0 or 2.1, but they can produce witnesses in format 1.0. To properly evaluate these verifiers, SV-COMP 2026 supports the format 1.0 in these cases. We emphasize that this *legacy support* is relevant only for inactive verifiers.

If there is no supported witness format or the only supported witness format is in demo mode, the witness generation is not required. Otherwise, a witness in some of the supported formats has to be generated. A witness in format 1.0 is written to file `witness.graphml`. A witness in format 2.0 or 2.1 is written to file `witness.yaml`. If a verifier generates both files, `witness.graphml` is ignored in the context of the verification track (it can be used in the validation track as explained below). We check whether (a) the generated witness is in a supported format, (b) its type corresponds to the verification result (correctness witnesses correspond to `TRUE`, violation witnesses to `FALSE`), and (c) it is syntactically valid. The last check is done by **WITNESSLINT** in case of C and by **PySVLIB-LINTER**^{new} in case of SV-LIB. We skip the last check in case of Java as there is no witness linter

available. If any of the check fails and a witness was required, the verification task is seen as unsolved by the verifier. The last sentence does not apply to the tasks of demo category *C.Huawei-Concurrency-Challenges* where witnesses are not considered important. Checked witnesses are turned into validation tasks. Competition scripts then run all suitable validators on each validation task, where suitability is determined by `category-structure.yml`. Each validator can either confirm the task, refute it, or fail to solve it. A validator confirms a correctness witness by returning TRUE and refutes it by returning FALSE. A violation witness is confirmed by FALSE and refuted by TRUE.

The validation track uses the same inputs, only the verification tasks are replaced by *validation tasks*. These are the handcrafted validation tasks in *C.ValidationCrafted* and all validation tasks generated by verifiers on the verification tasks in *C.Overall* that passed the checks mentioned above. If a verifier generated two witness files `witness.graphml` and `witness.yml` and both of them pass the checks, we consider both in the validation track. A validation task contains an expected result if it is handcrafted or if it was generated with an incorrect verification result (i.e., it contains a correctness witness for a verification task with property violation or a violation witness for a verification task with a correct program). In the latter case, the expected validation result corresponds to refutation. Each validator is executed on all validation tasks of the categories it participates in according to `category-structure.yml`.

Computing Resources. The computing resources for each tool execution were the same as in SV-COMP 2025 with the exception of shorter CPU time limit for validation tasks with correctness witnesses. Each verifier run was limited by 15 GB of memory and 15 min of cumulative CPU time on 4 processing units. Each run of a validator was limited to 2 processing units, 7 GB of memory, and 1.5 min of CPU time for violation witnesses and 5 min of CPU time for correctness witnesses. The time limit for validation of correctness witnesses was 15 min in the previous years. The change should motivate verifiers to produce witnesses whose validation is substantially easier than solving the original verification tasks. The machines for running the experiments are part of a computer cluster at the SoSy-Lab at LMU, which consists of 168 machines, where each machine has one Intel Xeon E3-1230 v5 CPU with 8 processing units, a frequency of 3.4 GHz, 33 GB of RAM, and a GNU/Linux operating system (x86_64-linux, Ubuntu 24.04 with Linux kernel 6.8). During the whole evaluation phase of the competition, approximately 1/3 of the cluster was not available due to a network switch failure. We used `BENCHEXEC` [45] to measure and control computing resources (CPU time, memory) and `BENCHCLOUD` [35] to distribute, install, run, and clean-up verification runs, and to collect the results.

Scoring Schema. For verification track, the scoring schema of SV-COMP 2026 in regular base categories was basically the one used since SV-COMP 2021. In Table 2 we list all the cases when verification results are awarded with non-zero points, where a verification result is

Table 2: Scores per individual verification results used since SV-COMP 2021

Result	Points	Description
TRUE correct	+2	Program correctly reported to satisfy the property and the correctness witness confirmed (or not required)
TRUE incorrect	-32	Incorrect program reported as correct (wrong proof)
FALSE correct	+1	Property violation was correctly found and the violation witness was confirmed
FALSE incorrect	-16	Violation reported but the property holds (false alarm)

- *incorrect* if it does not agree with the expected result of the verification task,
- *correct* if it agrees with the expected result and the generated witness is confirmed by some validator. In the categories where correctness witnesses are not required, a verification results is *correct* whenever it agrees with the expected result.

When a verification result agrees with the expected result but no validator confirms the witness (although it is required), we call the result *correct-unconfirmed*. As *C.FalseOverall* was designed to compare the bug-finding ability of individual verifiers, the points in its base categories are awarded only for results FALSE (both correct and incorrect), while the results TRUE are ignored. SV-COMP 2026 introduced a dual category called *C.TrueOverall* designed to compare the ability of verifiers to prove program correctness. Hence, the points in its base categories are awarded only for results TRUE, while the results FALSE are ignored.

Slightly different rules were applied in base demo categories. In the base categories of *C.Huawei-Concurrency-Challenges*, witnesses are not considered important and thus they are always treated as confirmed. In the base categories of *SV-LIB.Overall*, every correctness witness checked by the linter is immediately treated as confirmed because there was no validator for these witnesses available.

The score of a verifier in a base category is simply the sum of the points for individual verification tasks. The score for a meta category is computed from the scores of all contained (meta or base) categories on the next level and the number of tasks in these categories. Formally, if a meta category contains k categories of the next level and the i -th contained category has the score s_i and consists of n_i verification tasks, then the meta category gets the score $(\sum_{i=1}^k s_i/n_i) \cdot (\sum_{i=1}^k n_i)/k$, i.e., the sum of scores in each category normalized by the number of tasks in the category multiplied by the average number of tasks in the contained categories. Note that until SV-COMP 2024, the numbers n_i included also void tasks that are technically in categories but are not used in the competition (a verification task is marked as *void* typically because it was changed for some serious reason after the task freezing deadline). Since SV-COMP 2025, these void tasks are not included in n_i .

For validation track, the scoring schema of SV-COMP 2026 in base categories was the same as in the previous editions. The biggest difference from the verification track comes from the fact that many validation tasks do not contain the expected

Table 3: Scores per individual validation results used since SV-COMP 2024

Result	Points	Description for correctness witnesses Description for violation witnesses
TRUE correct	+2	The correctness witness was correctly confirmed The violation witness was correctly refuted
TRUE incorrect	-32	The correctness witness was confirmed but it is not correct The violation witness was refuted but it is correct
FALSE correct	+1	The correctness witness was correctly refuted The violation witness was correctly confirmed
FALSE incorrect	-16	The correctness witness was refuted but it is correct The violation witness was confirmed but it is not correct

validation result. In fact, it is contained only in handcrafted validation tasks and in the tasks generated by verifiers that solve some verification task incorrectly (and thus the witness they produce should be refuted). For the remaining tasks, the expected results are determined by voting. For each such a task, we collect the results from all validators that solved (i.e., confirmed or refuted) the task. If we have at least two such results and at least 75 % of them agree on their decision, then the expected result is set by the majority vote. In all other cases, the expected result is not determined and the validation task has no influence on the validation track results as it is considered *void*. Tool developers can inspect the results and convince the community that a voted expected results is in fact wrong. In such a case, the corresponding validation task is removed from the competition.

Table 3 lists all the cases when validation results are awarded with non-zero points. A validation result TRUE or FALSE is considered *correct* if it agrees with the expected result and *incorrect* otherwise. To compute the scores in base categories, we virtually divide each base category C into two subcategories C_c, C_w , where

- C_c contains the witnesses that are expected to be *correct* (i.e., correctness witnesses with the expected result TRUE and violation witnesses with the expected result FALSE), and
- C_w contains the witnesses that are expected to be *wrong* (i.e., correctness witnesses with the expected result FALSE and violation witnesses with the expected result TRUE).

The score of a validator in each subcategory C_c, C_w is the sum of the points for individual validation tasks in the subcategory. If some of the subcategories C_c, C_w is empty, the score for C is directly the score for the non-empty subcategory. If both subcategories C_c, C_w are non-empty and have respective scores s_c, s_w , then the score for C is computed as for a meta category in the verification track, i.e., $(\frac{s_c}{|C_c|} + \frac{s_w}{|C_w|}) \cdot \frac{|C_c| + |C_w|}{2}$. The score of a validator in a meta category is then computed by the same process as in the verification track. Note that all the scores presented in this paper and on the competition web are rounded to integers before printing, but their computation is done with a higher precision.

Table 4: Artifacts published for SV-COMP 2026

Content	DOI	Reference
Verification Tasks	10.5281/zenodo.18650775	[53]
Competition Results	10.5281/zenodo.18651757	[52]
Formal-Methods Tools	10.5281/zenodo.18650756	[33]
Verification Witnesses	10.5281/zenodo.18651735	[54]
Witness Format	10.5281/zenodo.17277275	[51]
BENCHEXEC	10.5281/zenodo.18455156	[151]
FM-WECK	10.5281/zenodo.18650812	[55]
BENCHCLOUD	10.5281/zenodo.18670174	[16]

Table 5: Publicly available components for reproducing SV-COMP 2026

Component	Repository	Version
Verification Tasks	gitlab.com/sosy-lab/benchmarking/sv-benchmarks	svcomp26
Benchmark Definitions	gitlab.com/sosy-lab/sv-comp/bench-defs	svcomp26
Formal-Methods Tools	gitlab.com/sosy-lab/benchmarking/fm-tools	2.3
BENCHEXEC (Benchmarking)	github.com/sosy-lab/benchexec	3.34
BENCHCLOUD (Distribution)	gitlab.com/sosy-lab/software/benchcloud	1.5.0
Witness Format	gitlab.com/sosy-lab/benchmarking/sv-witnesses	2.1.2
FM-WECK for CI	gitlab.com/sosy-lab/software/fm-weck	1.6.0
Processing Scripts	gitlab.com/sosy-lab/benchmarking/competition-scripts	svcomp26

Ranking and Medals. The rank of a verifier in each category was decided based on the achieved score. In case of a tie, we used the *success run time* as the secondary criterion, which is the total CPU time of the verifier over the tasks in a given category for which the verifier received positive points. Ranking in validation track works in the same way. We recall that inactive tools and *meta verifiers* (defined in the next section) are excluded from official rankings. In *C.Huawei-Concurrency-Challenges*, the tools with at least one Huawei employee in the team are excluded from the rankings too. In each meta category except demo categories and the meta categories contained in *C.FalseOverall* and *C.TrueOverall*, we assign medals to the three top-ranking tools with a positive score. Huawei provided prize money to the three top-ranking verifiers in the category *C.Huawei-Concurrency-Challenges*.

Reproducibility. SV-COMP results must be reproducible. Therefore, all competition artifacts are published at Zenodo (see [Table 4](#)) with the relevant tools and data to guarantee their long-term availability and immutability. Further, all major components are maintained in public version-control repositories. In [Table 5](#) we list these components with the links to the repositories and their versions used in SV-COMP 2026. Most of these components are described with more details in the SV-COMP 2016 report [24]. Later, [BENCHCLOUD](#) was introduced to distribute the benchmarking jobs in an elastic cloud and collect results. Moreover, [FM-WECK](#) was used to continuously check (GitLab CI pipeline)

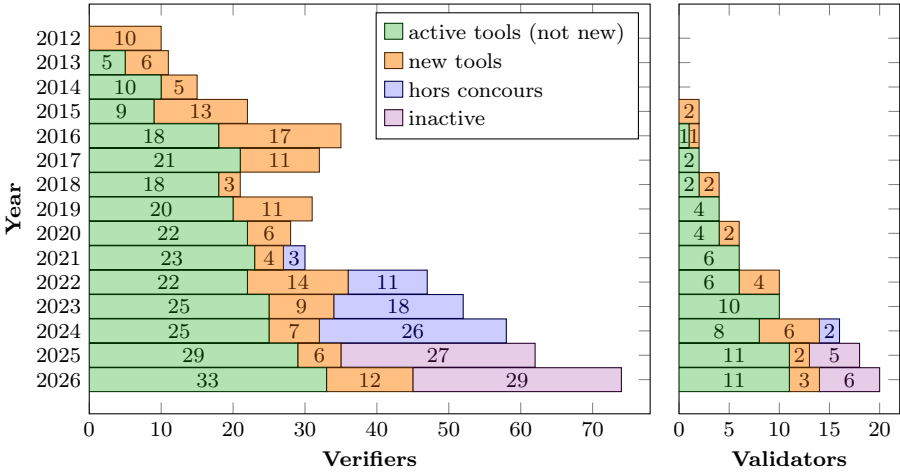


Fig. 3: Number of evaluated verifiers and validators in each year of SV-COMP; three new hors concours tools of 2022 counted only as new, not as hors concours

whether tools can be executed in the competition environment. The processing scripts to execute the experiments and post-process the data into tables, scores, and rankings are also publicly released.

For the reproducibility reasons, SV-COMP requires that the verifiers and validators must be publicly available for download and has a license that (a) allows reproduction and evaluation by anybody (including publication of results) and (b) does not restrict the usage of the verifier output (log files, witnesses). The verification and handcrafted validation tasks used in the competition are also accompanied by a license. In this case, the stated license must allow to

- view, understand, investigate, and reverse engineer the algorithm or system,
- change the program (in particular, pre-process and adopt the programs to be useful for a verification task),
- (re-)distribute the (original and changed) program under the same terms (in particular, in replication packages for research projects or as regression tests),
- compile and execute the program (in particular, for the purpose of verifying that a specification violation exists),
- and commercially take advantage of the program (in particular, to not exclude developers of commercial verifiers).

4 Participating Verifiers and Validators

In total, SV-COMP 2026 evaluates 74 verification and 20 validation tools. Besides 45 verifiers and 14 validators registered to and supported in the competition by development teams, we also evaluated some tools participating in previous years but not actively registered and supported this year (Fig. 3). These tools are called *inactive*, clearly marked with $\varnothing$ in all tables, and they do not appear in

rankings. Further, we clearly distinguish *meta verifiers* according to the following characterization approved by the community of SV-COMP 2023.

A *meta verifier* is a combination of at least two existing verification components such that each result produced by the combination can be computed by some of its components alone. A verifier is the result of research and engineering in verification algorithms and approaches, while a typical meta verifier selects a verification component to run, sets up its parameters, and potentially post-processes its output.

Meta verifiers are annotated with meta in all tables and also excluded from rankings. Note that before SV-COMP 2025, both inactive tools and meta verifiers were marked as *hors-concours* participants and not properly distinguished. Figure 3 shows the evolution of the number of verifiers and validators participating in individual editions of SV-COMP.

Tables 6 and 7 provide the list of all verifiers and validators evaluated in SV-COMP 2026, respectively. The tables contain the tool name (with hyperlink),

Table 6: Participating verifiers with tool references, representing jury members and their affiliations; ✗ for inactive, meta for meta verifiers, and new for first-time participants

Verifier	Ref.	Jury Member	Affiliation
2LS ✗	[59, 118]	–	–
AISE	[113, 150]	Z. Chen	NUDT, China
AProVE	[114]	N. Lommen	RWTH Aachen, Germany
BRICK	[60]	L. Bu	Nanjing U., China
BUBAAK ✗	[62, 64]	–	–
BUBAAK-SpLit ✗	[63]	–	–
CBMC ✗	[69, 108]	–	–
COASTAL ✗	[147]	–	–
CoOPERACE meta		V. Vojdani	U. Tartu, Estonia
CPACHECKER	[12, 13]	M. Jankola	LMU Munich, Germany
CPALockator ✗	[5, 6]	–	–
CPA-BAM-BnB ✗	[4, 149]	–	–
CPA-BAM-SMG ✗		–	–
CPV	[68]	P.-C. Chien	LMU Munich, Germany
CRUX ✗	[82, 141]	–	–
CSeq	[74, 103]	O. Inverso	Gran Sasso Sc. I., Italy
DARTAGNAN	[89, 112]	H. Ponce de León	Huawei Dresden, Germany
DASA new	[130]	F. Mächtle	U. Lübeck, Germany
DEAGLE	[93]	F. He	Tsinghua U., China
DIVINE ✗	[15, 109]	–	–
EBF ✗	[3]	–	–
EMERGENTHETA	[124, 125]	L. Bajczi	BME Budapest, Hungary
ESBMC-INCR	[70–73],	X. Li	U. Manchester, UK

(continues on next page)

Table 6: Participating verifiers (continued)

Verifier	Ref.	Jury Member	Affiliation
ESBMC-KIND	[88, 152]	X. Li	U. Manchester, UK
FRAMA-C-SV [⚡]	[47, 75]	–	–
GAZER-THETA [⚡]	[1, 92]	–	–
GDART	[128]	M. Mues	U. Wuppertal, Germany
GDART-LLVM [⚡]		–	–
GOBLINT	[140, 148]	S. Saan	U. Tartu, Estonia
GOBLITCH ^{new}	[97]	K. Holter	U. Tartu, Estonia
GRAVES-CPA ^{⚡ meta}	[110]	–	–
HORNIX	[57]	M. Blicha	U. Lugano, Switzerland
IEKKE ^{new}	[56]	P. Di Biase	Unimol, Italy
INFER [⚡]	[61, 106]	–	–
JAVA-RANGER [⚡]	[100, 143]	–	–
JAYHORN	[105, 142]	H. Mousavi	U. Tehran and TIAS, Iran
JBMC	[71, 72]	P. Schrammel	Diffblue, UK
JDART [⚡]	[117, 127]	–	–
JLiSA ^{new}	[7]	G. Zanatta	U. Venice, Italy
KORN	[85, 86]	G. Ernst	LMU Munich, Germany
LAZY-CSeq [⚡]	[101, 102]	–	–
LF-CHECKER [⚡]		–	–
LOCKSMITH [⚡]	[135]	–	–
MLB		L. Bu	Nanjing U., China
MOPSA	[104, 121]	R. Monat	Inria & U. Lille, France
MUVAL ^{new}		H. Unno	Tohoku U., Japan
NACPA ^{meta}	[111]	H. Wachowitz	LMU Munich, Germany
OGCHECKER ^{new}		Z. Yang	Xidian U., China
PeSCo-CPA ^{⚡ meta}	[137, 138]	–	–
PICHECKER [⚡]	[144]	–	–
PINAKA [⚡]	[66]	–	–
PREDATORHP [⚡]	[96, 133]	–	–
PROTON	[120, 129]	R. Metta	TCS, India
PySvLib-CHC ^{new}		G. Ernst	LMU Munich, Germany
RACERF	[76, 77]	T. Dacík	BUT, Czechia
ReFUNCTION ^{new}	[126]	N. Moussaoui Remil	Inria and ÉNS, France
Re3VER ^{new}	[155]	A. Štěpková	Masaryk U., Brno, Czechia
SEAL ^{new}	[58]	T. Dacík	Brno U. Techn., Czechia
SPF [⚡]	[131, 136]	–	–
SVF-SVC	[119]	M. Richards	U. New South Wales, AU
SvLibChecker ^{new}		M. Lingsch-Rosenfeld	LMU Munich, Germany
SV-SANITIZERS		S. Saan	U. Tartu, Estonia
SWAT	[115, 116]	N. Loose	U. Luebeck, Germany
SYMBIOTIC	[9, 65]	M. Jonáš	Masaryk U., Czechia
THETA	[145, 146]	C. Telbisz	BME Budapest, Hungary
THORN		L. Bajczi	BME Budapest, Hungary
UAUTOMIZER	[19, 94]	M. Heizmann	U. Freiburg, Germany
UGEMCUTTER	[87, 107]	D. Klumpp	U. Freiburg, Germany
UKOJAK	[84, 132]	M. Bentele	U. Freiburg, Germany

(continues on next page)

Table 6: Participating verifiers (continued)

Verifier	Ref.	Jury Member	Affiliation
UPARALIZER ^{new}	[17]	M. Barth	LMU Munich, Germany
UTAIPAN	[81, 91]	D. Dietsch	U. Freiburg, Germany
VERIAbs [∅]	[2, 78]	–	–
VERIAbsL [∅]	[79]	–	–
VERIOOVER [∅]	–	–	–

references to papers that describe the tool, the representing jury member and the affiliation. The listings are also available on the competition web site at <https://sv-comp.sosy-lab.org/2026/systems.php>. Table 7 additionally indicates witness formats and types supported by individual validators. The support of correctness witnesses in format 1.0 for C programs is indicated by ‘(✓)’ to emphasize that it is used in the competition only because of the legacy support of these

Table 7: Participating validators with tool references, representing jury members, their affiliations, and indication of relevant supported witnesses depending on format version and type; [∅] for inactive, ^{new} for first-time participants, *Cor.* for correctness witnesses, *Vio.* for violation witnesses, (✓) for legacy support, and ✓ for newly added support; validator **PySvLib-VALID.**^{new} supports violation witnesses in SV-LIB format

Validator	Ref.	Jury Member	Affiliation	Witness Format			
				1.0		2.0 or 2.1	
				Cor.	Vio.	Cor.	Vio.
CONCURRENTW2T	[14]	Z. Ádám	BME Budapest	✓			
CPACHECKER	[41, 44]	M. Lingsch-Rosenfeld	LMU Munich	(✓)	✓	✓	✓
CPA-w2T [∅]	[37, 39]	–	–		✓		
CProver-w2T [∅]	[37, 39]	–	–		✓		
DARTAGNAN	[134]	H. Ponce de León	Huawei Dresden		✓		
GOBLINT	[139]	S. Saan	U. Tartu				✓
GWIT [∅]	[99]	–	–		✓		
JCWIT [∅]	[67]	–	–	✓			
LIV	[48]	M. Lingsch-Rosenfeld	LMU Munich	(✓)		✓	
METAVAL	[46]	M. Lingsch-Rosenfeld	LMU Munich	(✓)	✓ [∅]	✓	✓
MOPSA	[123]	R. Monat	Inria & U. Lille			✓	
NITWIT [∅]	[156]	–	–		✓		
SYMBIOTIC-WITCH [∅]	[10]	–	–		✓		
THETA ^{new}		L. Bajczi	BME Budapest				✓
UAUTOMIZER	[36, 38]	M. Ebbinghaus	U. Freiburg	(✓)	✓	✓	✓
UGEMCUTTER ^{new}		D. Klumpp	U. Freiburg			✓	
UREFEREE		F. Schüssele	U. Freiburg	(✓)		✓	
Wit4JAVA	[153]	T. Wu	U. Manchester		✓		
WITCH	[8, 11]	P. Ayaziová	Masaryk U., Brno				✓
PYSVLIB-VALID. ^{new}		M. Lingsch-Rosenfeld	LMU Munich	(supports SV-LIB)			

witnesses. Note that to check these witnesses, we actually used the version of the validators participating in SV-COMP 2025 when these witnesses were still officially supported. Further, we used [METAVAL](#) in the version participating in SV-COMP 2025 to check violation witnesses in format 1.0 because in SV-COMP 2026, the tool was registered only to the other two validation subtracks. Hence, ‘✓[⊗]’ denotes that [METAVAL](#) participates in the corresponding subtrack as an inactive tool. [Table 8](#) lists the algorithms and techniques used by the verification and validation tools (some less frequent techniques were omitted due to limited space).

Table 8: Selected algorithms and techniques used by the participating tools; [⊗] for inactive, ^{meta} for meta verifiers, and ^{new} for first-time participants

Tool	Abstract Interpretation	Algorithm Selection	ARG-Based Analysis	Automata-Based Analysis	Bit-Precise Analysis	Bounded Model Checking	CEGAR	Concurrency Support	Constrained Horn Clauses	Explicit-Value Analysis	Floating-Point Arithmetics	Interpolation	k-Induction	Lazy Abstraction	Numeric Interval Analysis	Portfolio	Predicate Abstraction	Property-Directed Reach.	Ranking Functions	Separation Logic	Shape Analysis	Symbolic Execution	Targeted Input Generation	Task Translation
AISE																								
AProVE																								
BRICK						✓	✓	✓							✓				✓			✓		
BUBAAK [⊗]						✓		✓								✓			✓			✓		
BUBAAK-SpLit [⊗]		✓		✓	✓			✓					✓			✓			✓			✓		
CBMC [⊗]					✓	✓		✓											✓			✓		
COASTAL [⊗]																					✓			
CONCURRENTW2T								✓																
CoOpERACE ^{meta}		✓						✓								✓								
CPA-BAM-BnB [⊗]			✓		✓				✓	✓				✓										
CPACHECKER		✓	✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
CPALockATOR [⊗]			✓	✓	✓		✓	✓		✓		✓	✓	✓			✓	✓						
CPA-w2T [⊗]			✓	✓														✓						
CPV		✓			✓	✓	✓					✓	✓			✓	✓	✓						✓
CRUX [⊗]																					✓			
CSeq					✓	✓		✓																
DARTAGNAN					✓	✓		✓																
DASA ^{new}																						✓		
DEAGLE						✓		✓																
DIVINE [⊗]		✓			✓			✓	✓							✓					✓			
EBF [⊗]					✓	✓																		
EMERGENTHETA		✓	✓		✓	✓	✓	✓			✓	✓				✓	✓	✓						✓
ESBMC-INCR					✓	✓		✓					✓											
ESBMC-KIND					✓	✓		✓	✓	✓		✓			✓									
FRAMA-C-SV [⊗]															✓									
CProver-w2T [⊗]						✓																		

(continues on next page)

Table 8: Algorithms and techniques (continued)

Tool	Abstract Interpretation	Algorithm Selection	ARG-Based Analysis	Automata-Based Analysis	Bit-Precise Analysis	Bounded Model Checking	CEGAR	Concurrency Support	Constrained Horn Clauses	Explicit-Value Analysis	Floating-Point Arithmetics	Interpolation	k-Induction	Lazy Abstraction	Numeric Interval Analysis	Portfolio	Predicate Abstraction	Property-Directed Reach.	Ranking Functions	Separation Logic	Shape Analysis	Symbolic Execution	Targeted Input Generation	Task Translation
ReFuncTION ^{new}															✓									
GAZER-THETA [⊗]			✓				✓			✓		✓		✓		✓	✓							
GDART					✓	✓										✓								
GDART-LLVM [⊗]					✓	✓										✓						✓	✓	
GOBLINT	✓	✓			✓			✓		✓	✓				✓	✓								
GOBLINT-PAR		✓						✓			✓				✓	✓								
GOBLITCH ^{new}	✓	✓									✓				✓							✓		
GRAVES-CPA ^{⊗ meta}		✓	✓		✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓		✓		✓			
GWIT [⊗]					✓											✓						✓		
HORNIX									✓									✓						
IEKKE ^{new}						✓		✓																
INFER [⊗]															✓	✓				✓	✓			
JAVA-RANGER [⊗]					✓										✓	✓				✓		✓		
JAYHORN							✓		✓			✓		✓	✓		✓	✓				✓		
JBMC					✓	✓		✓																
JCWIT [⊗]						✓										✓						✓		
JDART [⊗]					✓											✓						✓		
JLiSA ^{new}	✓																							
KORN							✓		✓	✓						✓	✓					✓		
LAZY-CSEQ [⊗]					✓	✓		✓																
LIV																								✓
LOCKSMITH [⊗]								✓																
METAVAL			✓																					✓
MLB					✓												✓					✓		
MOPSA	✓	✓								✓	✓					✓	✓						✓	
MuVAL ^{new}																			✓					
NACPA ^{meta}		✓														✓								
NITWIT [⊗]										✓														
OGCHECKER ^{new}			✓					✓																
PeSCo-CPA ^{⊗ meta}		✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓		✓			
PICHECKER [⊗]		✓	✓		✓		✓	✓				✓					✓							
PINAKA [⊗]					✓	✓																✓		
PREDATORHP [⊗]																				✓				
PROTON						✓																		
PySvLIB-CHC ^{new}									✓															
RACERF								✓																
Re3VER ^{new}					✓			✓			✓		✓		✓	✓					✓	✓	✓	

(continues on next page)

Table 8: Algorithms and techniques (continued)

Tool	Abstract Interpretation	Algorithm Selection	ARG-Based Analysis	Automata-Based Analysis	Bit-Precise Analysis	Bounded Model Checking	CEGAR	Concurrency Support	Constrained Horn Clauses	Explicit-Value Analysis	Floating-Point Arithmetics	Interpolation	k-Induction	Lazy Abstraction	Numeric Interval Analysis	Portfolio	Predicate Abstraction	Property-Directed Reach.	Ranking Functions	Separation Logic	Shape Analysis	Symbolic Execution	Targeted Input Generation	Task Translation
SEAL ^{new}	✓																			✓	✓			
SPF [⊗]								✓													✓	✓		
SVF-SVC																						✓	✓	
SvLibChecker ^{new}			✓	✓		✓	✓		✓		✓	✓	✓	✓			✓					✓	✓	
SV-SANITIZERS								✓							✓							✓	✓	
SWAT																						✓	✓	
SYMBIOTIC					✓			✓		✓	✓	✓	✓		✓	✓	✓				✓	✓	✓	
SYMBIOTIC-WITCH [⊗]																					✓	✓		
THETA		✓	✓		✓	✓	✓	✓		✓	✓	✓	✓			✓	✓						✓	
THORN		✓							✓							✓	✓	✓						
2LS [⊗]					✓	✓							✓		✓				✓	✓	✓			
UAUTOMIZER		✓		✓	✓	✓	✓	✓				✓			✓		✓	✓		✓				
UGEMCUTTER		✓		✓	✓	✓	✓	✓				✓	✓			✓	✓	✓	✓					
UKOJAK					✓		✓	✓				✓	✓		✓		✓	✓						
UPARALIZER ^{new}				✓	✓	✓	✓	✓				✓	✓			✓	✓	✓						
UREFEREE		✓		✓	✓		✓	✓				✓	✓		✓		✓	✓	✓	✓				
UTAIPAN		✓		✓	✓		✓	✓		✓		✓	✓		✓		✓	✓		✓				
VERIABS [⊗]		✓					✓		✓	✓			✓		✓	✓	✓	✓						
VERIABSL [⊗]		✓				✓	✓		✓		✓		✓		✓	✓	✓	✓						
WITCH																					✓			
WIT4JAVA																						✓		

Table 9 gives an overview of common solver libraries and frameworks used by these tools. Note that both tables are based on information provided by teams that registered individual tools to SV-COMP 2026 or to some previous editions in the case of currently inactive tools. In Tables 8 and 9, we omitted the tools that would have no checkmark in the coresponding row. The web site <https://fm-tools.sosy-lab.org> provides more information about all tools evaluated in SV-COMP and Test-Comp since 2023 in a uniform way.

5 Results

The results of the competition represent the state of the art of what can be achieved with fully automatic software-verification tools on the given benchmark set. We report the *effectiveness* (the number of verification tasks that can be solved and correctness of the results, as accumulated in the score) and the *efficiency*

Table 9: Solver libraries and frameworks used as components by at least two participating tools;  for inactive,  for meta verifiers, and  for first-time participants

Tool	APRON	BITWUZLA	BOOLECTOR	CPACHECKER	CPROVER	CVC	ELDARICA	ESBMC	FRAMA-C	GLUCOSE	GOLEM	JAVASMT	JPF	KLEE	MATHSAT	MINISAT	SMTINTERPOL	SYMBIOTIC	ULTIMATE	Z3
AISE						✓														✓
APROVE (KoAT+LoAT)																				✓
BRICK																				✓
BUBAAK 																✓				✓
CBMC 					✓												✓			✓
COASTAL 													✓							
CPA-BAM-BnB 				✓								✓			✓					
CPA-BAM-SMG 				✓								✓			✓					
CPACHECKER	✓			✓								✓			✓					
CPALOCKATOR 				✓								✓			✓					
CPV		✓	✓			✓									✓	✓				
CRUX 																				✓
CSEQ					✓											✓				
DARTAGNAN												✓								
DEAGLE																				
EBF 								✓							✓					
EMERGENTHETA		✓	✓			✓						✓			✓		✓			✓
ESBMC-INCR								✓							✓					
ESBMC-KIND								✓							✓					
FRAMA-C-SV 									✓											
REFUNCTION 	✓																			
GDART						✓														✓
GDART-LLVM 																				✓
GOBLINT	✓																			
GOBLINT-PAR	✓																			
GOBLITCH 	✓																			
GRAVES-CPA  				✓								✓			✓			✓		✓
HORNIX																				
IEKKE 																✓				
JAVA-RANGER 													✓							
JBMC					✓					✓						✓				✓
JDART 						✓							✓							✓
KORN							✓				✓									✓
LAZY-CSEQ 					✓											✓				
MOPSA	✓																			
MUVAL 																				✓
NACPA 				✓																
OGCHECKER 				✓																
PESCO-CPA  				✓								✓			✓					
PICHECKER 				✓								✓			✓		✓			

(continues on next page)

Table 9: Solver libraries and frameworks (continued)

Tool	APRON	BITWUZLA	BOOLECTOR	CPACHECKER	CPROVER	CVC	ELDARICA	ESBMC	FRAMA-C	GLUCOSE	GOLEM	JAVASMT	JPF	KLEE	MATHSAT	MINISAT	SMTINTERPOL	SYMBIOTIC	ULTIMATE	Z3
PYSVLIB-CHC ^{new}						✓					✓									✓
RACERF								✓												
RE3VER ^{new}														✓				✓		✓
SEAL ^{new}								✓												
SPF [⊗]												✓								
SvLIBCHECKER ^{new}															✓					✓
SWAT																				✓
SYMBIOTIC															✓			✓		✓
THETA		✓	✓			✓						✓			✓		✓			✓
2LS [⊗]					✓					✓						✓				✓
UAUTOMIZER						✓									✓		✓		✓	✓
UGEMCUTTER						✓									✓		✓		✓	✓
UKOJAK															✓		✓		✓	✓
UPARALIZER ^{new}						✓									✓		✓		✓	✓
UREFEREE						✓									✓		✓		✓	✓
UTAI PAN						✓									✓		✓		✓	✓
VERIABS [⊗]				✓	✓											✓				✓
VERIABSL [⊗]				✓	✓											✓				✓
WITCH													✓					✓		✓

(resource consumption in terms of CPU time). The results are presented in the same way as in last years, such that the improvements compared to the last years are easy to identify. The results presented in this report were provided to the participants in advance and their objections have been settled.

Consumed Resources. Before we present the competition results, we report some statistics to give an impression of the overall computation work. One complete execution of all verifiers in the competition consisted of 1 069 940 verification runs (each verifier runs on each verification task of the categories in which the verifier participates), consuming 9 years of CPU time (without validation). Witness validation required 24 million validation runs (each validator runs on each validation task of the categories in which the validator participates) consuming 8 years of CPU time. During the training phase of the competition, we executed 7.5 million verification runs consuming 53 years of CPU time and 95 milion validation runs consuming 18 years of CPU time.

Verification track. Tables 10, 11, and 12 present the quantitative overview of all verifiers in all meta categories except the subcategories of *C.FalseOverall* and *C.TrueOverall*. We split the presentation into three tables, one for the verifiers of C programs that participate with an active team support (Table 10), one for the inactive verifiers of C programs (Table 11), and the last for all verifiers of Java programs and SV-LIB tasks (Table 12). The head row lists meta categories

Table 10: Overview of the results of all actively participating verifiers for C; empty cells indicate opt-outs, meta for meta verifiers, new for first-time participants

Verifier	C.ReachSafety 14375 tasks max. score 23331	C.MemSafety 4145 tasks max. score 6529	C.Concurrency 3124 tasks max. score 5656	C.NoOverflows 8218 tasks max. score 13281	C.Termination 2146 tasks max. score 3734	C.SoftwareSystems 4394 tasks max. score 7140	C.FalseOverall 36402 tasks max. score 12195	C.TrueOverall 36402 tasks max. score 48413	C.Overall 36402 tasks max. score 60609	C.Huawei-Con.-Chall. 66 tasks max. score 123
AISE										
APROVE					2077					
BRICK										
CoOPERACE ^{meta}										
CPACHECKER	12866	4646	2146	8474	1152	1496	7389	20589	27979	3
CPV	10626				1358					
CSeq			-12677							-48
DARTAGNAN			3516							71
DEAGLE			4630							-11
EMERGENTheta	4414	642	2636	1821	796	-18	1091	10400	11491	0
ESBMC-INCR			2435							-70
ESBMC-KIND	11037	3309	2410	9109	1074	14	5232	18732	23964	-70
GOBLINT	4664	2840	2555	8658	1614	457		22673	22673	34
GOBLITCH ^{new}	6552					611				
HORNIX										
IEKKE ^{new}			3428							-12
KORN										
MOPSA	4501	3165	0	9906	1571	2746	1442	20637	22079	0
MUVAL ^{new}					356					
NACPA ^{meta}	5959	3940	1578	7869	860	1477	4089	17538	21627	1
OGCHECKER ^{new}			131							0
PROTON					3368					
RACERF										
ReFUNCTION ^{new}					294					
Re3VER ^{new}	2583									
SEAI ^{new}										
SV-SANITIZERS		878		837						
SVF-SVC	-9866	0				-346				
SYMBIOTIC	10788	4738	42	8476	1427	2614	7833	17638	25471	0
Theta	5252	695	2157	2586	796	-18	-329	11885	11556	0
THORN	738	12	1247	-174	583	69	187	4178	4365	0
UAUTOMIZER	8243	3957	2986	10831	2616	730	5835	25637	31472	0
UGEMCUTTER			3189							1
UKOJAK	6012	2982	0	8821	0	290	3468	10347	13815	0
UPARALIZER ^{new}	8279			11049						
UTAIPAN	7013	3517	2453	10645	0	404	4243	17044	21287	0

Table 11: Overview of the results of inactive verifiers for C; empty cells indicate opt-outs, $\emptyset$ for inactive, *meta* for meta verifiers

Verifier	C.ReachSafety 14375 tasks max. score 23331	C.MemSafety 4145 tasks max. score 6529	C.Concurrency 3124 tasks max. score 5656	C.NoOverflows 8218 tasks max. score 13281	C.Termination 2146 tasks max. score 3734	C.SoftwareSystems 4394 tasks max. score 7140	C.FalseOverall 36402 tasks max. score 12195	C.TrueOverall 36402 tasks max. score 48413	C.Overall 36402 tasks max. score 60609	C.Huawei-Con.-Chall. 66 tasks max. score 123
2LS $\emptyset$	8100	631	0	6808	1607	25	3063	10883	13946	0
BUBAAK $\emptyset$	9071	3296	-191	6483	1332	2121	6362	13401	19763	0
BUBAAK-SPLIT $\emptyset$	7959	3283	-191	6472	1042	1996	6192	12083	18275	0
CBMC $\emptyset$	1896	1817	812	7173	1085	-1690	-2932	13998	11066	0
CPA-BAM-BnB $\emptyset$						-1577				
CPA-BAM-SMG $\emptyset$		3039				-3361				
CPALockATOR $\emptyset$			-3540							0
CRUX $\emptyset$	2821			602						
DIVINE $\emptyset$	6371	532	379	0	0	-2524	476	242	718	2
EBF $\emptyset$			351							-88
FRAMA-C-SV $\emptyset$				1579						
GAZER-THETA $\emptyset$										
GDART-LLVM $\emptyset$										
GRAVES-CPA $\emptyset$ <i>meta</i>	4252	2106	238	3329	-994	-362	-6603	11090	4487	-35
INFER $\emptyset$	-131447		-8297	-77663		-28819				-354
LAZY-CSEQ $\emptyset$			-14183							-48
LF-CHECKER $\emptyset$			440							-18
LOCKSMITH $\emptyset$										
PeSCO-CPA $\emptyset$ <i>meta</i>	8085	2904	1483	8424	985	-3112	2414	12836	15249	-15
PICHECKER $\emptyset$			382							-18
PINAKA $\emptyset$	3818			641	878					
PREDATORHP $\emptyset$		4543								
VERIAbs $\emptyset$	14056									
VERIAbsL $\emptyset$	14404									
VERIOOVER $\emptyset$										

with the number of valid tasks and the maximal score for each category. The tools are listed in alphabetical order (in Table 12, verifiers for Java are before those for SV-LIB) and every table row lists the scores of one verifier. An empty table cell means that the verifier did not participate in the corresponding meta category. In Tables 10 and 12, we indicate the top three candidates in each (non-demo) category by formatting their scores in bold face and in larger font size. We recall that inactive tools and meta verifiers are excluded from rankings. More information (including interactive tables, quantile plots for every category,

Table 12: Overview of the results of all verifiers for Java and SV-LIB; $\varnothing$ for inactive, meta for meta verifiers, new for first-time participants

Verifier	Java.Overall	SV-LIB.Overall demo
	1731 tasks max. score 2821	261 tasks max. score 391
COASTAL $\varnothing$	-13809	
DASA new	210	
GDART	1470	
JAVA-RANGER $\varnothing$	1059	
JAYHORN	-2472	
JBMC	1561	
JDART $\varnothing$	-11255	
JLISA new	1311	
MLB	335	
SPF $\varnothing$	-4741	
SWAT	1291	
CPACHECKER		133
PySVLIB-CHC new		78
SvLIBCHECKER new		165

and also the raw data in XML format) is available on the competition web site (<https://sv-comp.sosy-lab.org/2026/results>) and in the results artifact (see Table 4). Note that the results for subcategories of *C.FalseOverall* and *C.TrueOverall* are not explicitly presented neither in this report nor on the web due to their marginal significance. The results can be obtained from the detailed results of the corresponding categories in *C.Overall* presented on the web.

Table 13 shows the medalists for each (non-demo) meta category. The cumulative run time of the verifier on these tasks is presented in the column ‘CPU Time’. The column ‘Solved Tasks’ shows the number of tasks in the corresponding category for which the verifier received positive points. The column ‘Unconf. Tasks’ provides the number of tasks for which the verifier returned a correct answer, but the corresponding required witness was not confirmed by any validator. The columns ‘False Alarms’ and ‘Wrong Proofs’ provide the number of verification tasks for which the verifier returned incorrect FALSE and incorrect TRUE, respectively. Recall that categories *C.FalseOverall* and *C.TrueOverall* completely ignore TRUE and FALSE results, respectively. As a consequence, there cannot be any wrong proof in *C.FalseOverall* or false alarm in *C.TrueOverall*.

The results in *C.Huawei-Concurrency-Challenges* are available in Tables 10 and 11. Because the representing jury member of DARTAGNAN is a Huawei employee and DIVINE $\varnothing$ is inactive, the prize money went to GOBLINT (3 000 EUR), CPACHECKER (1 500 EUR), and UGEMCUTTER (500 EUR).

Score-Based Quantile Functions. The community often present the results of the comparative evaluation in the form of score-based quantile functions [21, 45] because these visualizations are easy to understand and provide more infor-

Table 13: Verification: Overview of the medalists in each (non-demo) meta category; values for ‘CPU time’ in hours and rounded to two significant digits, ‘Solved Tasks’ shows the number of tasks for which the verifier got positive points, ‘Unconf. Tasks’ shows the number of tasks where the verifier returned the correct answer and the corresponding required witness was not confirmed

<i>Category</i>	<i>Rank</i>	<i>Verifier</i>	<i>Score</i>	<i>CPU Time</i>	<i>Solved Tasks</i>	<i>Unconf. Tasks</i>	<i>False Alarms</i>	<i>Wrong Proofs</i>
<i>C.ReachSafety</i> (14375 tasks, max. score 23331)								
1		CPACHECKER	12866	150	8 053	365	2	0
2		ESBMC-KIND	11037	57	7 273	2 456	0	0
3		SYMBIOTIC	10788	110	4 080	207	1	0
<i>C.MemSafety</i> (4145 tasks, max. score 6529)								
1		SYMBIOTIC	4738	3.9	3 834	11	0	2
2		CPACHECKER	4646	19	3 814	5	1	0
3		UAUTOMIZER	3957	35	2 226	80	0	0
<i>C.Concurrency</i> (3124 tasks, max. score 5656)								
1		DEAGLE	4630	3.1	2 513	19	1	3
2		DARTAGNAN	3516	19	2 052	43	3	1
3		IEKKE ^{new}	3428	2.3	2 473	216	44	22
<i>C.NoOverflows</i> (8218 tasks, max. score 13281)								
1		UPARALIZER ^{new}	11049	88	6 700	22	0	0
2		UAUTOMIZER	10831	67	6 669	59	1	0
3		UTAIPAN	10645	75	6 572	56	0	0
<i>C.Termination</i> (2146 tasks, max. score 3734)								
1		PROTON	3368	22	1 780	162	1	0
2		UAUTOMIZER	2616	17	1 558	5	0	0
3		AProVE	2077	31	1 011	42	0	0
<i>C.SoftwareSystems</i> (4394 tasks, max. score 7140)								
1		MOPSA	2746	11	1 920	28	0	0
2		SYMBIOTIC	2614	6.2	1 165	94	1	0
3		CPACHECKER	1496	28	1 857	257	0	0
<i>C.FalseOverall</i> (36402 tasks, max. score 12195)								
1		SYMBIOTIC	7833	42	7 378	52	2	
2		CPACHECKER	7389	100	7 668	454	4	
3		UAUTOMIZER	5835	53	4 864	569	7	
<i>C.TrueOverall</i> (36402 tasks, max. score 48413)								
1		UAUTOMIZER	25637	200	11 686	167		3
2		GOBLINT	22673	24	10 364	352		0
3		MOPSA	20637	26	9 766	26		0
<i>C.Overall</i> (36402 tasks, max. score 60609)								
1		UAUTOMIZER	31472	260	16 550	736	7	3
2		CPACHECKER	27979	240	21 632	930	4	0
3		SYMBIOTIC	25471	160	16 158	347	2	3
<i>Java.Overall</i> (1731 tasks, max. score 2821)								
1		JBMC	1561	1.5	1 033	315	1	13
2		GDART	1470	4.3	920	107	1	1
3		JLiSA ^{new}	1311	2.0	617	1	0	0

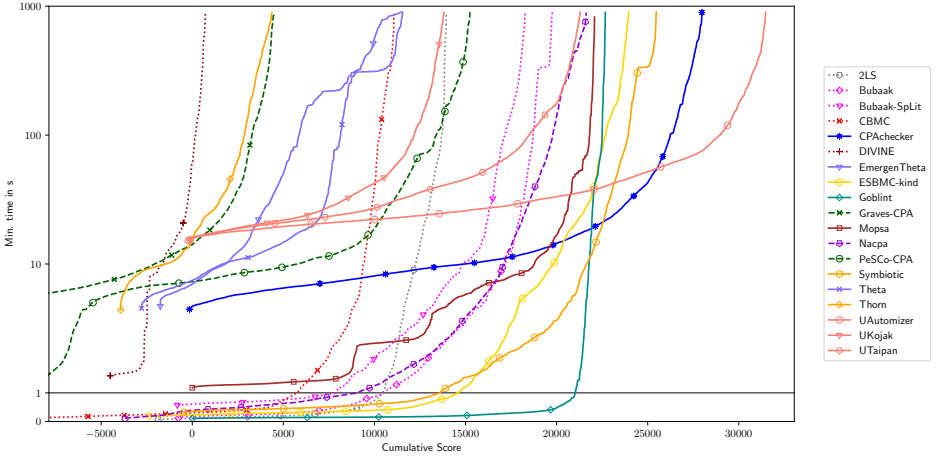


Fig. 4: Quantile functions for category *C.Overall*; each quantile function illustrates the quantile (x -coordinate) of the scores obtained by correct verification runs below a certain run time (y -coordinate), minus the overall penalty for incorrect results; more details were given previously [21]; a logarithmic scale is used for the time range from 1 s to 1000 s, and a linear scale is used for the time range between 0 s and 1 s

mation than simple tables. The results archive (see Table 4) and the web site (<https://sv-comp.sosy-lab.org/2026/results>) include such a plot for each category. As an example, we show the plot for category *C.Overall* in Fig. 4. A total number of 19 verifiers participated in this category. Since SV-COMP 2025, we use different line styles to distinguish regular (i.e., not meta verifiers) active participants (solid line), active meta verifiers (dashed line), and all inactive tools (dotted line). For each participant, the quantile plot shows the overall performance in *C.Overall*. The plot says that the winner of *C.Overall* is **UAUTOMIZER** as its graph ends furthest to the right. The graph for **UAUTOMIZER** also starts slightly left from $x = 0$ because the verifier produced 10 incorrect results and therefore received some negative points. Also other verifiers whose graphs start with a negative cumulative score produced incorrect results. The graphs also illustrate run-time characteristics: The area below a graph is roughly proportional to the accumulated run time. It is easy to see that **GOBLINT** performs best regarding run time. A more detailed discussion of score-based quantile plots, including examples of what insights one can obtain from the plots, is provided in previous competition reports [21, 24].

Validation Track. Validation of verification witnesses was pioneered by SV-COMP in 2015. Shortly after that, verification witnesses become more and more important for various reasons: they do not only justify and help to understand and interpret verification results, but they also serve as exchange object for intermediate results and allow to make use of imprecise verification techniques (e.g., via machine learning). However, a case study on the quality of the results of witness validators [49] published in 2022 revealed a great potential for improvements. To

Table 14: Validation of correctness witnesses in formats 2.0 and 2.1: Overview of the medalists in each meta category; CPU time in hours and rounded to two significant digits

<i>Category</i>	<i>Rank</i>	<i>Validator</i>	<i>Score</i>	<i>CPU Time</i>	<i>Solved Tasks</i>	<i>Wrong Confirm.</i>	<i>Wrong Refut.</i>
<i>C.ReachSafety</i> (42807 tasks, max. score 64211)							
	1	METAVAL	51786	310	38 102	17	0
	2	UAUTOMIZER	35695	290	17 806	3	0
	3	CPACHECKER	28910	250	31 346	0	16
<i>C.NoOverflows</i> (47221 tasks, max. score 70832)							
	1	UAUTOMIZER	63546	380	43 643	1	0
	2	CPACHECKER	54068	110	36 689	2	0
	3	GOBLINT	39671	2.4	34 788	0	0
<i>C.SoftwareSystems</i> (15708 tasks, max. score 21991)							
	1	GOBLINT	9411	25	11 856	0	0
	2	UAUTOMIZER	7706	140	12 596	6	0
	3	CPACHECKER	4314	20	5 354	0	0
<i>C.ValidationCrafted</i> (18 tasks, max. score 27)							
	1	MOPSA	14	0.0080	12	0	0
	2	GOBLINT	13	0.0010	11	0	0
	3	METAVAL	13	0.068	16	1	0
<i>C.Overall</i> (105754 tasks, max. score 155987)							
	1	UAUTOMIZER	89983	810	74 061	11	0
	2	CPACHECKER	71251	380	73 398	2	16
	3	GOBLINT	65792	36	52 863	0	0

stimulate further advances in verification witnesses and their validators, the study suggested that witness validators should also undergo a periodical comparative evaluation and proposed a scoring schema for witness-validation results. This materializes in the validation track for C programs run by SV-COMP since 2023.

Thanks to the development and adoption of the witness format 2.0 [8], we could eliminate the use of correctness witnesses in format 1.0 from the validation track. Hence, the track has now three subtracks:

1. validation of correctness witnesses in formats 2.0 and 2.1 (see Table 14),
2. validation of violation witnesses in formats 2.0 and 2.1 (see Table 15), and
3. validation of violation witnesses in format 1.0 (see Table 16).

The tables present only the medalists in non-empty meta categories in each subtrack. If some category has less than 3 medalists, it means that less than three validators with active team support reached a positive score. Two non-empty meta categories of the last subtrack, namely *C.NoOverflows* and *C.Overall*, are completely missing in Table 16 as neither of the two validators with active team support participating in these categories reached a positive score. The

Table 15: Validation of violation witnesses in formats 2.0 and 2.1: Overview of the medalists in each meta category; CPU time in hours and rounded to two significant digits

<i>Category</i>	<i>Rank</i>	<i>Validator</i>	<i>Score</i>	<i>CPU Time</i>	<i>Solved Tasks</i>	<i>Wrong Confirm.</i>	<i>Wrong Refut.</i>
<i>C.ReachSafety</i> (19560 tasks, max. score 30039)							
1		CPACHECKER	16829	50	12 312	4	131
2		WITCH	16204	12	7 605	0	24
3		UAUTOMIZER	8477	69	8 497	1	158
<i>C.MemSafety</i> (655 tasks, max. score 901)							
1		UAUTOMIZER	832	4.8	558	0	0
2		WITCH	791	0.50	653	0	0
3		CPACHECKER	632	1.2	650	0	0
<i>C.NoOverflows</i> (9664 tasks, max. score 12080)							
1		UAUTOMIZER	11196	56	9 625	1	0
2		CPACHECKER	9961	20	9 237	0	0
3		WITCH	9147	9.0	9 112	0	0
<i>C.Termination</i> (1483 tasks, max. score 1483)							
1		WITCH	1359	0.39	1 298	0	0
2		CPACHECKER	1348	12	1 205	0	0
3		UAUTOMIZER	1238	13	1 426	0	0
<i>C.SoftwareSystems</i> (824 tasks, max. score 1288)							
1		WITCH	759	0.62	446	0	0
2		UAUTOMIZER	566	3.1	321	0	0
3		CPACHECKER	524	2.1	455	0	0
<i>C.ValidationCrafted</i> (117 tasks, max. score 176)							
1		WITCH	163	0.041	115	0	0
2		THETA ^{new}	24	0.013	5	0	0
<i>C.Overall</i> (32303 tasks, max. score 44272)							
1		WITCH	33434	22	19 229	0	24
2		CPACHECKER	18482	86	23 916	8	131
3		THETA ^{new}	4665	2.9	539	0	0

column ‘Wrong Confirm.’ gives the number of cases when the respective validator confirmed a witness that was incorrect. The column ‘Wrong Refut.’ shows the number of cases when the respective validator refuted a correct witness. We recall that the correctness or incorrectness of many witnesses used in the validation track is determined by voting and thus the numbers of wrong confirmations and refutations do not have to be completely objective.

The complete results of all validators in all relevant categories of all subtracks are available in the results artifact (see Table 4) and on the SV-COMP web site (<https://sv-comp.sosy-lab.org/2026/results/results-validated/>).

The limited support of some properties and program features by the witness formats (see Table 1), missing medalists in some categories, and the negative scores in the detailed results on the web site: all of these show the need of further research and development in the area of software verification witnesses and their validation.

Table 16: Validation of violation witnesses in format 1.0: Overview of the medalists in each meta category; CPU time in hours and rounded to two significant digits

Category	Rank	Validator	Score	CPU Time	Solved Tasks	Wrong Confirm.	Wrong Refut.
<i>C.ReachSafety</i> (35069 tasks, max. score 53856)							
	1	CONCURRENTW2T	3389	1.7	2 784	26	0
<i>C.MemSafety</i> (9390 tasks, max. score 14085)							
	1	CPACHECKER	6653	19	9 181	21	28
<i>C.Concurrency</i> (3799 tasks, max. score 5699)							
	1	DARTAGNAN	3299	7.9	1 795	14	0
	2	UAUTOMIZER	3199	23	3 268	3	45
	3	CPACHECKER	1487	3.6	800	24	11
<i>C.Termination</i> (1685 tasks, max. score 2528)							
	1	UAUTOMIZER	1211	12	1 661	0	0
<i>C.SoftwareSystems</i> (7003 tasks, max. score 12255)							
	1	CPACHECKER	3893	16	2 962	15	12
	2	UAUTOMIZER	1795	45	4 929	3	338

Table 16 indicates diminishing interest of validator developers in supporting the old witness format 1.0. As a consequence, SV-COMP will abandon this format in the future.

6 Conclusion

The 15th edition of the Competition on Software Verification (SV-COMP 2026) compared 74 automatic tools for software verification (including 12 new ones and 29 tools without active team support) and 20 automatic tools for validation of verification witnesses (including 3 new and 6 tools without active team support). The overall numbers of evaluated verifiers and validators were historically the highest (see Fig. 3). The total number of verification tasks with C or Java programs significantly increased to precisely 38 133. On top of that, the competition considered 254 verification tasks in the recently introduced format SV-LIB for software-verification tasks.

The results of the competition show good progress in both verification and witness validation, especially in the adoption and further development of the witness format 2.0. However, even the best verification and validation tools still produce some incorrect results and are far from solving all benchmarks. For example, category *C.ReachSafety* offers 14 375 verification tasks (max. score 23 331), but the best verifier in that category can solve only 8 053 tasks (score 12 866). This motivates further improvements of verifiers. Similar situation holds for validators. For example, out of the 19 560 validation tasks with violation witnesses 2.0 or 2.1 in category *C.ReachSafety* (max. score 30 039), the best validator in that category solved 12 312 tasks (score 16 829). Finally, the competition motivates further extensions of the witness format to support more properties and program features.

SV-COMP received remarkable support from outside its own community. A research group developed the tool ARG-V [122], in order to extend the benchmark set for Java from real software projects found on GitHub. Intel open-sourced their security-critical firmware component Intel® TDX Module and allocated resources to develop and contribute an industrial benchmark set [34]. The TACAS conference acknowledged the importance of benchmarks and comparative evaluations by accepting for publication in this TACAS 2026 proceedings volume the two above works, this report, and 15 competition contributions. Huawei provided travel support to enable researchers to participate in SV-COMP, awarded prizes for a dedicated demo category of challenges in concurrency verification, and supported the infrastructure of the competition.

Data-Availability Statement. The verification tasks and results of the competition are published at Zenodo, as described in Table 4. All components and data that are necessary for reproducing the competition are available in public version repositories, as specified in Table 5. For easy access, the results are presented also online on the competition web site <https://sv-comp.sosy-lab.org/2026>.

Funding Statement. SV-COMP 2026 was supported by Huawei – Dresden Research Center, Germany. Some participants of this competition were funded in part by the Deutsche Forschungsgemeinschaft (DFG) — 378803395 (ConVeY) and 536040111 (Bridge). Jan Strejček has been supported by the Czech Science Foundation grant GA23-06506S.

Acknowledgments. We thank the organization committee for their help in running the competition and for their work on improving the quality of the verification and validation tasks, the jury for their advice in refining and application of the competition rules and for their work in the paper selection process, and the verification community for contributing their tools to the evaluation and submitting new benchmarks. Furthermore, we thank Schloss Dagstuhl for hosting a seminar that contributed to the development of witness and exchange formats [42] and TACAS for hosting SV-COMP as part of their program at ETAPS.

References

1. Ádám, Zs., Sallai, Gy., Hajdu, Á.: GAZER-THETA: LLVM-based verifier portfolio with BMC/CEGAR (competition contribution). In: Proc. TACAS (2). pp. 433–437. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_27
2. Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: VERIABS: Verification by abstraction and test generation. In: Proc. ASE. pp. 1138–1141. IEEE (2019). <https://doi.org/10.1109/ASE.2019.00121>
3. Aljaafari, F., Shmarov, F., Manino, E., Menezes, R., Cordeiro, L.: EBF 4.2: Black-Box cooperative verification for concurrent programs (competition contribution). In: Proc. TACAS (2). pp. 541–546. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_33

4. Andrianov, P., Friedberger, K., Mandrykin, M.U., Mutilin, V.S., Volkov, A.: CPA-BAM-BNB: Block-abstraction memoization and region-based memory models for predicate abstractions (competition contribution). In: Proc. TACAS (2). pp. 355–359. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_22
5. Andrianov, P., Mutilin, V., Khoroshilov, A.: CPALockator: Thread-modular approach with projections (competition contribution). In: Proc. TACAS (2). pp. 423–427. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_25
6. Andrianov, P.S.: Analysis of correct synchronization of operating system components. Program. Comput. Softw. **46**, 712–730 (2020). <https://doi.org/10.1134/S0361768820080022>
7. Arceri, V., Negrini, L., Zanatta, G., Bianchi, F., Lisovenko, T., Olivieri, L., Ferrara, P.: JLiSA: The Java frontend of the library for static analysis (competition contribution). In: Proc. TACAS (2). pp. 543–550. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_30
8. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Proc. SPIN. pp. 184–203. LNCS 14624, Springer (2024). https://doi.org/10.1007/978-3-031-66149-5_11
9. Ayaziová, P., Jonáš, M., Mihalkovič, V., Sedláček, J., Strejček, J.: SYMBIOTIC 11: Predicate abstraction joins the party (competition contribution). In: Proc. TACAS (2). pp. 583–588. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_36
10. Ayaziová, P., Strejček, J.: SYMBIOTIC-WITCH 2: More efficient algorithm and witness refutation (competition contribution). In: Proc. TACAS (2). pp. 523–528. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_30
11. Ayaziová, P., Strejček, J.: WITCH 3: Validation of violation witnesses in the witness format 2.0 (competition contribution). In: Proc. TACAS (3). pp. 341–346. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_18
12. Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Software verification with CPACHECKER 3.0: Tutorial and user guide. In: Proc. FM. pp. 543–570. LNCS 14934, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_30
13. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPACHECKER 2.3 with strategy selection (competition contribution). In: Proc. TACAS (3). pp. 359–364. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
14. Bajczi, L., Ádám, Zs., Micskei, Z.: CONCURRENTWITNESS2TEST: Test-harnessing the power of concurrency (competition contribution). In: Proc. TACAS (3). pp. 330–334. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_16
15. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkal, P., Štill, V.: Model checking of C and C++ with DIVINE 4. In: Proc. ATVA. pp. 201–207. LNCS 10482, Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_14
16. Barth, M., Beyer, D., Chien, P.C., Jankola, M.: Benchcloud Release 1.5.0. Zenodo (2026). <https://doi.org/10.5281/zenodo.18670174>
17. Barth, M., Dietsch, D., Heizmann, M., Jakobs, M.-C.: ULTIMATE PARALIZER: Parallel trace abstraction (competition contribution). In: Proc. TACAS (2). pp. 595–599. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_38
18. Bartocci, E., Beyer, D., Black, P.E., Fedyukovich, G., Garavel, H., Hartmanns, A., Huisman, M., Kordon, F., Nagele, J., Sighireanu, M., Steffen, B., Suda, M., Sutcliffe, G., Weber, T., Yamada, A.: TOOLympics 2019: An overview of competitions in formal methods. In: Proc. TACAS (3). pp. 3–24. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_1

19. Bentele, M., Barth, M., Ebbinghaus, M., Körner, J., Dietsch, D., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: *ULTIMATE AUTOMIZER with a one-dimensional memory model (competition contribution)*. In: Proc. TACAS (2). pp. 589–594. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_37
20. Beyer, D.: *Competition on software verification (SV-COMP)*. In: Proc. TACAS. pp. 504–524. LNCS 7214, Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_38
21. Beyer, D.: *Second competition on software verification (Summary of SV-COMP 2013)*. In: Proc. TACAS. pp. 594–609. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_43
22. Beyer, D.: *Status report on software verification (Competition summary SV-COMP 2014)*. In: Proc. TACAS. pp. 373–388. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_25
23. Beyer, D.: *Software verification and verifiable witnesses (Report on SV-COMP 2015)*. In: Proc. TACAS. pp. 401–416. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_31
24. Beyer, D.: *Reliable and reproducible competition results with BENCHEXEC and witnesses (Report on SV-COMP 2016)*. In: Proc. TACAS. pp. 887–904. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_55
25. Beyer, D.: *Software verification with validation of results (Report on SV-COMP 2017)*. In: Proc. TACAS (2). pp. 331–349. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_20
26. Beyer, D.: *Automatic verification of C and Java programs: SV-COMP 2019*. In: Proc. TACAS (3). pp. 133–155. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_9
27. Beyer, D.: *Advances in automatic software verification: SV-COMP 2020*. In: Proc. TACAS (2). pp. 347–367. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_21
28. Beyer, D.: *Software verification: 10th comparative evaluation (SV-COMP 2021)*. In: Proc. TACAS (2). pp. 401–422. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_24
29. Beyer, D.: *Progress on software verification: SV-COMP 2022*. In: Proc. TACAS (2). pp. 375–402. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20
30. Beyer, D.: *Competition on software verification and witness validation: SV-COMP 2023*. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
31. Beyer, D.: *State of the art in software verification and witness validation: SV-COMP 2024*. In: Proc. TACAS (3). pp. 299–329. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_15
32. Beyer, D.: *Evaluating tools for automatic software testing (Report on Test-Comp 2026)*. In: Proc. FASE. pp. 449–468. LNCS 16504, Springer (2026). https://doi.org/10.1007/978-3-032-22774-4_23
33. Beyer, D.: *FM-Tools Release 2.3: Data set of metadata about tools for formal methods (SV-COMP 2026, Test-Comp 2026)*. Zenodo (2026). <https://doi.org/10.5281/zenodo.18650756>
34. Beyer, D., Chien, P.C., Huang, B.Y., Lee, N.Z., Lemberger, T.: *A case study in firmware verification: Applying formal methods to Intel[®] TDX Module*. In: Proc. TACAS (2). pp. 42–64. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_3

35. Beyer, D., Chien, P.C., Jankola, M.: BENCHCLOUD: A platform for scalable performance benchmarking. In: Proc. ASE. pp. 2386–2389. ACM (2024). <https://doi.org/10.1145/3691620.3695358>
36. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). <https://doi.org/10.1145/2950290.2950351>
37. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). <https://doi.org/10.1145/3477579>
38. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). <https://doi.org/10.1145/2786805.2786867>
39. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
40. Beyer, D., Ernst, G., Jonáš, M., Lingsch-Rosenfeld, M.: SV-LIB 1.0: A standard exchange format for software-verification tasks. arXiv/CoRR **2511**(21509) (December 2025). <https://doi.org/10.48550/arXiv.2511.21509>
41. Beyer, D., Friedberger, K.: Violation witnesses and result validation for multi-threaded programs. In: Proc. ISOla (1). pp. 449–470. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_26
42. Beyer, D., Huisman, M., Strejček, J., Wehrheim, H.: Information exchange in software verification (Dagstuhl Seminar 25172). Dagstuhl Reports **15**(4), 92–111 (2025). <https://doi.org/10.4230/DAGREP.15.4.92>
43. Beyer, D., Jankola, M., Lingsch-Rosenfeld, M.: Transition invariants revisited: Termination witnesses and their validation (2025)
44. Beyer, D., Lingsch-Rosenfeld, M.: CPACHECKER 4.0 as witness validator (competition contribution). In: Proc. TACAS (3). pp. 192–198. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_11
45. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). <https://doi.org/10.1007/s10009-017-0469-y>
46. Beyer, D., Spiessl, M.: METAVAL: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10
47. Beyer, D., Spiessl, M.: The static analyzer FRAMA-C in SV-COMP (competition contribution). In: Proc. TACAS (2). pp. 429–434. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_26
48. Beyer, D., Spiessl, M.: LIV: A loop-invariant validation using straight-line programs. In: Proc. ASE. pp. 2074–2077. IEEE (2023). <https://doi.org/10.1109/ASE56229.2023.00214>
49. Beyer, D., Strejček, J.: Case study on verification-witness validators: Where we are and where we go. In: Proc. SAS. pp. 160–174. LNCS 13790, Springer (2022). https://doi.org/10.1007/978-3-031-22308-2_8
50. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS (3). pp. 151–186. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_9
51. Beyer, D., Strejček, J.: SV-Witnesses – Format 2.1. Zenodo (2025). <https://doi.org/10.5281/zenodo.17277275>
52. Beyer, D., Strejček, J.: Results of the 15th Intl. Competition on Software Verification (SV-COMP 2026). Zenodo (2026). <https://doi.org/10.5281/zenodo.18651757>

53. Beyer, D., Strejček, J.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2026, Test-Comp 2026). Zenodo (2026). <https://doi.org/10.5281/zenodo.18650775>
54. Beyer, D., Strejček, J.: Verification witnesses from verification tools (SV-COMP 2026). Zenodo (2026). <https://doi.org/10.5281/zenodo.18651735>
55. Beyer, D., Wachowitz, H.: FM-WECK Release 1.6.0. Zenodo (2026). <https://doi.org/10.5281/zenodo.18650812>
56. Biase, P.D., Fischer, B., Torre, S.L., Schrammel, P., Parlato, G.: IEKKE: A SAT-based bounded-round verifier for multi-threaded programs (competition contribution). In: Proc. TACAS (2). pp. 536–542. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_29
57. Blichla, M., Kofroň, J., Glitta, O.: HORNIX: From LLVM IR to constrained Horn clauses and back (competition contribution). In: Proc. TACAS (2). pp. 530–535. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_28
58. Brablec, T., Dacík, T., Vojnar, T.: SEAL: Symbolic execution with separation logic (competition contribution). In: Proc. TACAS (2). pp. 571–576. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_34
59. Brain, M., Joshi, S., Kröning, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Proc. SAS. pp. 145–161. LNCS 9291, Springer (2015). https://doi.org/10.1007/978-3-662-48288-9_9
60. Bu, L., Xie, Z., Lyu, L., Li, Y., Guo, X., Zhao, J., Li, X.: BRICK: Path enumeration-based bounded reachability checking of C programs (competition contribution). In: Proc. TACAS (2). pp. 408–412. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_22
61. Calcagno, C., Distefano, D., O’Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (2011). <https://doi.org/10.1145/2049697.2049700>
62. Chalupa, M., Henzinger, T.: BUBAAK: Runtime monitoring of program verifiers (competition contribution). In: Proc. TACAS (2). pp. 535–540. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_32
63. Chalupa, M., Richter, C.: BUBAAK-SPLit: Split what you cannot verify (competition contribution). In: Proc. TACAS (3). pp. 353–358. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_20
64. Chalupa, M., Richter, C.: BUBAAK: Dynamic cooperative verification (competition contribution). In: Proc. TACAS (3). pp. 212–216. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_14
65. Chalupa, M., Strejček, J., Vitovská, M.: Joint forces for memory safety checking. In: Proc. SPIN. pp. 115–132. Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_7
66. Chaudhary, E., Joshi, S.: PINAKA: Symbolic execution meets incremental solving (competition contribution). In: Proc. TACAS (3). pp. 234–238. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_20
67. Cheng, Z., Wu, T., Schrammel, P., Tihanyi, N., de Lima Filho, E.B., Cordeiro, L.C.: JCWIT: A correctness-witness validator for Java programs based on bounded model checking. In: Proc. ISSTA. pp. 1831–1835. ACM (2024). <https://doi.org/10.1145/3650212.3685303>
68. Chien, P.C., Lee, N.Z.: CPV: A circuit-based program verifier (competition contribution). In: Proc. TACAS (3). pp. 365–370. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_22

69. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
70. Cordeiro, L.C., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: Proc. ICSE. pp. 331–340. ACM (2011). <https://doi.org/10.1145/1985793.1985839>
71. Cordeiro, L.C., Kesseli, P., Kröning, D., Schrammel, P., Trtík, M.: JBMC: A bounded model checking tool for verifying Java bytecode. In: Proc. CAV. pp. 183–190. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_10
72. Cordeiro, L.C., Kröning, D., Schrammel, P.: JBMC: Bounded model checking for Java bytecode (competition contribution). In: Proc. TACAS (3). pp. 219–223. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_17
73. Cordeiro, L.C., Morse, J., Nicole, D., Fischer, B.: Context-bounded model checking with ESBMC 1.17 (competition contribution). In: Proc. TACAS. pp. 534–537. LNCS 7214, Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_42
74. Coto, A., Inverso, O., Sales, E., Tuosto, E.: A prototype for data race detection in CSeq 3 (competition contribution). In: Proc. TACAS (2). pp. 413–417. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_23
75. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: FRAMA-C. In: Proc. SEFM. pp. 233–247. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16
76. Dacík, T., Vojnar, T.: Racerf: Lightweight static data race detection for C code (experience paper). In: Proc. ECOOP. pp. 37:1–37:19. LIPIcs 333, Schloss Dagstuhl (2025). <https://doi.org/10.4230/LIPIcs.ECOOP.2025.37>
77. Dacík, T., Vojnar, T.: RACERF: Data race detection with Frama-C (competition contribution). In: Proc. TACAS (3). pp. 248–253. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_20
78. Darke, P., Agrawal, S., Venkatesh, R.: VERIABS: A tool for scalable verification by abstraction (competition contribution). In: Proc. TACAS (2). pp. 458–462. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_32
79. Darke, P., Chimdyalwar, B., Agrawal, S., Venkatesh, R., Chakraborty, S., Kumar, S.: VERIABSL: Scalable verification by abstraction and strategy prediction (competition contribution). In: Proc. TACAS (2). pp. 588–593. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_41
80. Denis, X., Siegel, S.F.: VerifyThis 2023: An international program verification competition. In: TOOLympics Challenge 2023 - Updates, Results, Successes of the Formal-Methods Competitions, TOOLympics 2023, Paris, France. pp. 147–159. LNCS 14550, Springer (2023). https://doi.org/10.1007/978-3-031-67695-6_5
81. Dietsch, D., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: ULTIMATE TAIPAN 2023 (competition contribution). In: Proc. TACAS (2). pp. 582–587. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_40
82. Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D., Tomb, A.: Constructing semantic models of programs with the software analysis workbench. In: Proc. VSTTE. pp. 56–72. LNCS 9971, Springer (2016). https://doi.org/10.1007/978-3-319-48869-1_5
83. Erhard, J., Bentele, M., Heizmann, M., Klumpp, D., Saan, S., Schüssele, F., Schwarz, M., Seidl, H., Tilscher, S., Vojdani, V.: Correctness witnesses for concurrent programs: Bridging the semantic divide with ghosts. In: Proc. VMCAI, Part I. pp. 74–100. LNCS 15529, Springer (2025). https://doi.org/10.1007/978-3-031-82700-6_4

84. Ermis, E., Hoenicke, J., Podelski, A.: Splitting via interpolants. In: Proc. VMAI. pp. 186–201. LNCS 7148, Springer (2012). https://doi.org/10.1007/978-3-642-27940-9_13
85. Ernst, G.: A complete approach to loop verification with invariants and summaries. Tech. Rep. arXiv:2010.05812v2, arXiv (January 2020). <https://doi.org/10.48550/arXiv.2010.05812>
86. Ernst, G.: KORN: Horn clause based verification of C programs (competition contribution). In: Proc. TACAS (2). pp. 559–564. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_36
87. Farzan, A., Klumpp, D., Podelski, A.: Sound sequentialization for concurrent program verification. In: Proc. PLDI. pp. 506–521. ACM (2022). <https://doi.org/10.1145/3519939.3523727>
88. Gadelha, M.Y., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via k -induction. Int. J. Softw. Tools Technol. Transf. **19**(1), 97–114 (February 2017). <https://doi.org/10.1007/s10009-015-0407-9>
89. Gavrilenko, N., Ponce de León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC for weak memory models: Relation analysis for compact SMT encodings. In: Proc. CAV. pp. 355–365. LNCS 11561, Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_19
90. Giesl, J., Mesnard, F., Rubio, A., Thiemann, R., Waldmann, J.: Termination competition (termCOMP 2015). In: Proc. CADE. pp. 105–108. LNCS 9195, Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_6
91. Greitschus, M., Dietsch, D., Podelski, A.: Loop invariants from counterexamples. In: Proc. SAS. pp. 128–147. LNCS 10422, Springer (2017). https://doi.org/10.1007/978-3-319-66706-5_7
92. Hajdu, Á., Micskei, Z.: Efficient strategies for CEGAR-based model checking. J. Autom. Reasoning **64**(6), 1051–1091 (2020). <https://doi.org/10.1007/s10817-019-09535-x>
93. He, F., Sun, Z., Fan, H.: DEAGLE: An SMT-based verifier for multi-threaded programs (competition contribution). In: Proc. TACAS (2). pp. 424–428. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_25
94. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
95. Heizmann, M., Klumpp, D., Lingsch-Rosenfeld, M., Schüssele, F.: Correctness witnesses with function contracts. arXiv/CoRR **2501**(12313) (January 2025). <https://doi.org/10.48550/arXiv.2501.12313>
96. Holík, L., Kotoun, M., Peringer, P., Šoková, V., Trtík, M., Vojnar, T.: PREDATOR shape analysis tool suite. In: Hardware and Software: Verification and Testing. pp. 202–209. LNCS 10028, Springer (2016). <https://doi.org/10.1007/978-3-319-49052-6>
97. Holter, K., Ayaziová, P., Saan, S., Strejček, J., Vojdani, V.: GOBLITCH: Combining abstract interpretation with symbolic execution via witnesses (competition contribution). In: Proc. TACAS (2). pp. 523–529. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_27
98. Howar, F., Jasper, M., Mues, M., Schmidt, D.A., Steffen, B.: The RERS challenge: Towards controllable and scalable benchmark synthesis. Int. J. Softw. Tools Technol. Transf. **23**(6), 917–930 (2021). <https://doi.org/10.1007/s10009-021-00617-z>
99. Howar, F., Mues, M.: GWIT (competition contribution). In: Proc. TACAS (2). pp. 446–450. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_29

100. Hussein, S., Yan, Q., McCamant, S., Sharma, V., Whalen, M.: JAVA RANGER: Supporting string and array operations (competition contribution). In: Proc. TACAS (2). pp. 553–558. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_35
101. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: LAZY-CSEQ: A lazy sequentialization tool for C (competition contribution). In: Proc. TACAS. pp. 398–401. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_29
102. Inverso, O., Tomasco, E., Fischer, B., Torre, S.L., Parlato, G.: Bounded verification of multi-threaded programs via lazy sequentialization. *ACM Trans. Program. Lang. Syst.* **44**(1), 1:1–1:50 (2022). <https://doi.org/10.1145/3478536>
103. Inverso, O., Trubiani, C.: Parallel and distributed bounded model checking of multi-threaded programs. In: Proc. PPOPP. pp. 202–216. ACM (2020). <https://doi.org/10.1145/3332466.3374529>
104. Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: Proc. VSTTE. pp. 1–18. LNCS 12031, Springer (2019). https://doi.org/10.1007/978-3-030-41600-3_1
105. Kahsai, T., Rümmer, P., Sanchez, H., Schäfer, M.: JAYHORN: A framework for verifying Java programs. In: Proc. CAV. pp. 352–358. LNCS 9779, Springer (2016). https://doi.org/10.1007/978-3-319-41528-4_19
106. Kettl, M., Lemberger, T.: The static analyzer INFER in SV-COMP (competition contribution). In: Proc. TACAS (2). pp. 451–456. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_30
107. Klumpp, D., Dietsch, D., Heizmann, M., Schüssele, F., Ebbinghaus, M., Farzan, A., Podolski, A.: ULTIMATE GEMCUTTER and the axes of generalization (competition contribution). In: Proc. TACAS (2). pp. 479–483. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_35
108. Kröning, D., Tautschnig, M.: CBMC: C bounded model checker (competition contribution). In: Proc. TACAS. pp. 389–391. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26
109. Lauko, H., Ročkal, P., Barnat, J.: Symbolic computation via program transformation. In: Proc. ICTAC. pp. 313–332. Springer (2018). https://doi.org/10.1007/978-3-030-02508-3_17
110. Leeson, W., Dwyer, M.: GRAVES-CPA: A graph-attention verifier selector (competition contribution). In: Proc. TACAS (2). pp. 440–445. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_28
111. Lemberger, T., Wachowitz, H.: NACPA: Native checking with parallel-portfolio analyses (competition contribution). In: Proc. TACAS (3). pp. 236–241. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_18
112. de Leon, H.P., Furbach, F., Heljanko, K., Meyer, R.: DARTAGNAN: Bounded model checking for weak memory models (competition contribution). In: Proc. TACAS (2). pp. 378–382. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_24
113. Lin, Y., Chen, Z., Wang, J.: AISE v2.0: Combining loop transformations (competition contribution). In: Proc. TACAS (3). pp. 199–204. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_12
114. Lommen, N., Giesl, J.: APROVE (KoAT+LoAT) (competition contribution). In: Proc. TACAS (3). pp. 205–211. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_13
115. Loose, N., Mächtle, F., Sieck, F., Eisenbarth, T.: SWAT: Modular dynamic symbolic execution for Java applications using dynamic instrumentation (competition

- contribution). In: Proc. TACAS (3). pp. 399–405. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_28
116. Loose, N., Sieck, F., Mächtle, F., Eisenbarth, T.: SWAT: Improvements to the symbolic executor (competition contribution). In: Proc. TACAS (2). pp. 577–582. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_35
117. Luckow, K.S., Dimjasevic, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamaric, Z., Raman, V.: JDART: A dynamic symbolic analysis framework. In: Proc. TACAS. pp. 442–459. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_26
118. Malík, V., Schrammel, P., Vojnar, T., Nečas, F.: 2LS: Arrays and loop unwinding (competition contribution). In: Proc. TACAS (2). pp. 529–534. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_31
119. McGowan, C., Richards, M., Sui, Y.: SVF-SVC: Software verification using SVF (competition contribution). In: Proc. TACAS (3). pp. 254–259. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_21
120. Metta, R., Karmarkar, H., Madhukar, K., Venkatesh, R., Chakraborty, S.: PROTON: Probes for non-termination and termination (competition contribution). In: Proc. TACAS (3). pp. 393–398. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_27
121. Milanese, M., Monat, R., Ouadjaout, A., Miné, A.: MOPSA-C: Towards incorrectness and termination verdicts (competition contribution). In: Proc. TACAS (2). pp. 551–558. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_31
122. Moloney, C., Dyer, R., Sherman, E.: Demonstrating ARG-V’s generation of realistic Java benchmarks for SV-COMP. In: Proc. TACAS (2). pp. 277–285. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_14
123. Monat, R., Milanese, M., Parolini, F., Boillot, J., Ouadjaout, A., Miné, A.: MOPSA-C: Improved verification for C programs, simple validation of correctness witnesses (competition contribution). In: Proc. TACAS (3). pp. 387–392. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_26
124. Mondok, M., Bajczi, L., Szekeres, D., Molnár, V.: EMERGENTHETA: Variations on symbolic transition systems (competition contribution). In: Proc. TACAS (3). pp. 217–222. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_15
125. Mondok, M., Telbisz, C., Bajczi, L., Kovács, D., Dobos-Kovács, M., Molnár, V.: EMERGENTHETA: Experimental analyses within the theta framework (competition contribution). In: Proc. TACAS (2). pp. 510–515. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_25
126. Moussaoui-Remil, N., Urban, C.: REFUNCTION: Conditional termination by abstract interpretation of numerical C programs (competition contribution). In: Proc. TACAS (2). pp. 565–570. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_33
127. Mues, M., Howar, F.: JDART: Portfolio solving, breadth-first search and SMT-Lib strings (competition contribution). In: Proc. TACAS (2). pp. 448–452. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_30
128. Mues, M., Howar, F.: GDART (competition contribution). In: Proc. TACAS (2). pp. 435–439. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_27
129. Mukhopadhyay, D., Metta, R., Karmarkar, H., Madhukar, K.: PROTON 2.1: Synthesizing ranking functions via fine-tuned locally hosted LLM (competition contribution). In: Proc. TACAS (3). pp. 242–247. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_19

130. Mächtle, F., Serr, J.-N., Loose, N., Eisenbarth, T.: DASA: Fully gradient-based program analysis (competition contribution). In: Proc. TACAS (2). pp. 503–509. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_24
131. Noller, Y., Păsăreanu, C.S., Le, X.B.D., Visser, W., Fromherz, A.: Symbolic PATHFINDER for SV-COMP (competition contribution). In: Proc. TACAS (3). pp. 239–243. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_21
132. Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: ULTIMATE KOJAK with memory safety checks (competition contribution). In: Proc. TACAS. pp. 458–460. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_44
133. Peringer, P., Šoková, V., Vojnar, T.: PREDATORHP revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: Proc. TACAS (2). pp. 408–412. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_30
134. Ponce-De-Leon, H., Haas, T., Meyer, R.: DARTAGNAN: SMT-based violation witness validation (competition contribution). In: Proc. TACAS (2). pp. 418–423. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_24
135. Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Practical static race detection for C. ACM Trans. Program. Lang. Syst. **33**(1) (January 2011). <https://doi.org/10.1145/1889997.1890000>
136. Păsăreanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehltz, P.C., Rungta, N.: Symbolic PATHFINDER: Integrating symbolic execution with model checking for Java bytecode analysis. Autom. Software Eng. **20**(3), 391–425 (2013). <https://doi.org/10.1007/s10515-013-0122-2>
137. Richter, C., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). <https://doi.org/10.1007/s10515-020-00270-x>
138. Richter, C., Wehrheim, H.: PESCO: Predicting sequential combinations of verifiers (competition contribution). In: Proc. TACAS (3). pp. 229–233. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_19
139. Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: GOBLINT VALIDATOR: Correctness witness validation by abstract interpretation (competition contribution). In: Proc. TACAS (3). pp. 335–340. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_17
140. Saan, S., Kocal, A.R., Petter, M., Holter, K., Erhard, J., Schwarz, M., Vojdani, V., Seidl, H.: GOBLINT: A portfolio for mixed flow-sensitive abstract interpretation (competition contribution). In: Proc. TACAS (2). pp. 516–522. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_26
141. Scott, R., Dockins, R., Ravitch, T., Tomb, A.: CRUX: Symbolic execution meets SMT-based verification (competition contribution). Zenodo (February 2022). <https://doi.org/10.5281/zenodo.6147218>
142. Shamakhi, A., Hojjat, H., Rümmer, P.: Towards string support in JAYHORN (competition contribution). In: Proc. TACAS (2). pp. 443–447. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_29
143. Sharma, V., Hussein, S., Whalen, M.W., McCamant, S.A., Visser, W.: JAVA RANGER: Statically summarizing regions for efficient symbolic execution of Java. In: Proc. ESEC/FSE. pp. 123–134. ACM (2020). <https://doi.org/10.1145/3368089.3409734>
144. Su, J., Yang, Z., Xing, H., Yang, J., Tian, C., Duan, Z.: PICHECKER: A POR and interpolation-based verifier for concurrent programs (competition contri-

- bution). In: Proc. TACAS (2). pp. 571–576. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_38
145. Telbisz, C., Bajczi, L., Szekeres, D., Vörös, A.: THETA: Various approaches for concurrent program verification (competition contribution). In: Proc. TACAS (3). pp. 260–265. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_22
 146. Tóth, T., Hajdu, A., Vörös, A., Micskei, Z., Majzik, I.: THETA: A framework for abstraction refinement-based model checking. In: Proc. FMCAD. pp. 176–179 (2017). <https://doi.org/10.23919/FMCAD.2017.8102257>
 147. Visser, W., Geldenhuys, J.: COASTAL: Combining concolic and fuzzing for Java (competition contribution). In: Proc. TACAS (2). pp. 373–377. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_23
 148. Vojdani, V., Apinis, K., Rötov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: The Goblint approach. In: Proc. ASE. pp. 391–402. ACM (2016). <https://doi.org/10.1145/2970276.2970337>
 149. Volkov, A.R., Mandrykin, M.U.: Predicate abstractions memory modeling method with separation into disjoint regions. Proceedings of the Institute for System Programming (ISPRAS) **29**, 203–216 (2017). [https://doi.org/10.15514/ISPRAS-2017-29\(4\)-13](https://doi.org/10.15514/ISPRAS-2017-29(4)-13)
 150. Wang, Z., Chen, Z.: AISE: A symbolic verifier by synergizing abstract interpretation and symbolic execution (competition contribution). In: Proc. TACAS (3). pp. 347–352. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_19
 151. Wendler, P., Beyer, D.: sosy-lab/benchexec: Release 3.34. Zenodo (2026). <https://doi.org/10.5281/zenodo.18455156>
 152. Wu, T., Li, X., Manino, E., Menezes, R., Gadelha, M., Xiong, S., Tihanyi, N., Petoumenos, P., Cordeiro, L.: ESBMC v7.7: Efficient concurrent software verification with scheduling, incremental SMT and partial order reduction (competition contribution). In: Proc. TACAS (3). pp. 223–228. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_16
 153. Wu, T., Schrammel, P., Cordeiro, L.: WIT4JAVA: A violation-witness validator for Java verifiers (competition contribution). In: Proc. TACAS (2). pp. 484–489. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_36
 154. Ádám, Zs., Ayaziová, P., Bajczi, L., Beyer, D., Jankola, M., Lingsch-Rosenfeld, M., Strejček, J.: Non-termination witnesses and their validation. In: Proc. ASE 2025. pp. 1969–1981. IEEE (2025). <https://doi.org/10.1109/ASE63991.2025.00164>
 155. Štěpková, A., Jonáš, M., Strejček, J.: RE3VER: Reverse and verify (competition contribution). In: Proc. TACAS (2). pp. 559–564. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_32
 156. J. Švejda, Berger, P., Katoen, J.P.: Interpretation-based violation witness validation for C: NITWIT. In: Proc. TACAS. pp. 40–57. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_3

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





DASA: Fully Gradient-Based Program Analysis (Competition Contribution)

Felix Mächtle^{*} , Jan-Niclas Serr , Nils Loose , and Thomas Eisenbarth

Institute for IT Security, University of Lübeck, Lübeck, Germany
{f.maehtle,j.serr,n.loose,thomas.eisenbarth}@uni-luebeck.de

Abstract. We present a novel approach to program analysis that models each instruction using a mathematically differentiable wrapper. This enables gradient-based optimization over program inputs to drive execution towards specific target states, such as assertion violations. Instead of enumerating execution paths or solving symbolic constraints, the program itself is transformed into a differentiable computation graph. By optimizing a loss function that represents the likelihood of reaching a target location, all execution paths are explored simultaneously within a single optimization process.

This paper describes the design and implementation of DASA, a differentiable analysis framework for Java programs, and reports on its first participation in the Java category of SV-COMP 2026.

Keywords: Dynamic Symbolic Execution · Gradient-Based Optimization · Java Verification · Differentiable Programming · SV-COMP 2026

1 Verification Approach

Program analysis traditionally relies on symbolic execution [7] or fuzzing [26] to discover program states such as assertion violations. Fuzzing explores the input space using random or mutation-based strategies, which scale well but struggle with deep or complex control-flow conditions. Symbolic execution, in contrast, reasons about program paths using symbolic values and constraint solvers, enabling precise reasoning about program semantics. However, symbolic execution suffers from limitations such as path explosion due to branching or solver timeouts on complex arithmetic.

To address these challenges, several learning-based approaches have been proposed. Neural program smoothing, exemplified by NEUZZ [24], learns differentiable surrogate models to enable gradient-based input generation, but has been shown to generalize poorly beyond shallow program behaviors [19]. Gradient-guided fuzzers such as Angora [4] use taint analysis and byte-level gradients to solve branch conditions more efficiently, yet still explore discrete execution paths. Other methods treat programs as black boxes and apply input-space optimization [3], but lack direct access to gradients of program semantics, relying instead on heuristic guidance rather than end-to-end differentiation.

^{*} Jury member

Original program

```

def test(i):
    i += 5
    if i > 20:
        print("big")
    else:
        print("small")

```

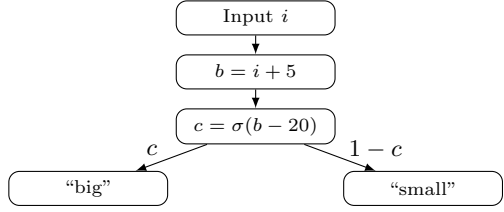


Fig. 1. Transformation of a simple program into a differentiable computation graph. Minimizing the loss $\mathcal{L} = -\sigma(i + 5 - 20)$ maximizes the likelihood of executing the “then” branch with input i .

DASA (Differentiable Abstract Symbolic Analyzer), presented in this work, draws inspiration from neural program analysis [24] and gradient-based SMT solving [9], but differs significantly in scope and implementation. DASA constructs a fully differentiable representation of the program itself. This enables end-to-end gradient-based optimization. Every instruction is replaced by a mathematically differentiable counterpart. Arithmetic operations map directly to differentiable operators, comparisons are approximated using sigmoid functions (σ), and strings are represented using continuous vector encodings whose character distributions are produced via softmax functions. DASA also differentiates control flow. The program is represented as a computation graph, where each node corresponds to an instruction and edges encode data and control dependencies. Branching instructions propagate continuous control-flow scores derived from the evaluated branch condition. As a result, all program paths are explored simultaneously within a single differentiable graph. By defining a loss function that maximizes the control-flow score of a target instruction (e.g., an assertion failure), gradient-based optimization can directly relate program inputs to the likelihood of executing the target instruction.

Figure 1 illustrates this concept for a simple example. A short program is converted into a differentiable computation graph, where branch conditions are smoothed and translated into a loss function. Optimizing this loss yields input values that drive execution towards the desired program state.

2 System Architecture

DASA transforms Java programs into differentiable computation graphs and applies gradient-based optimization to discover assertion violations. The architecture consists of the following five main components:

Graph Extraction. DASA interfaces with GraalVM [21] to extract a compiler intermediate representation (IR) of the target program. This IR encodes all operations, control flow, and data dependencies in a graph-based form. Consequently, DASA benefits from GraalVM’s advanced graph optimizations, such

as partial loop unrolling and code simplification. Across the SV-COMP benchmarks, GraalVM generates a total of 271 unique node types, averaging 158.7 nodes per task. For SV-COMP, the graph extraction takes on average around 280 seconds CPU time, accounting for 2/3 of the total computation time of the cases, in which DASA found a violation.

Differentiable Execution. The extracted IR graph is mapped to a differentiable computation graph implemented in PyTorch [22]. Each IR node is replaced by a corresponding differentiable operator that approximates its semantics. The current implementation of DASA includes 46 types of nodes, ranging from control-flow implementations, over numeric operations to basic support for complex types including Strings and Arrays. Arithmetic operations map directly to PyTorch operators, while comparisons are smoothed using sigmoid functions with an annealing factor that gradually increases, allowing the optimizer to explore broadly early on and converge to discrete decisions later.

Optimization. DASA performs gradient-based optimization over the program inputs. A loss function is defined to maximize the control-flow probability of reaching a target location and triggering an assertion. Candidate inputs obtained through optimization are validated by concrete execution.

Seed Initialization. Gradient-based optimization requires initial values for all input variables. Therefore, DASA employs a lightweight, type-aware seeding strategy that combines domain knowledge with randomized sampling: For each input, initial values are drawn from a mixture of (i) commonly error-prone boundary values (e.g., 0, -1, MIN_VALUE, MAX_VALUE), (ii) constants extracted from the program, and (iii) random samples from type-specific distributions, inspired by AutoStub [12]. To mitigate convergence to local optima, DASA performs multiple optimization restarts using independently sampled seeds.

Witness Creation. DASA relies on the SWAT [11] witness generator to produce witnesses. The optimized control-flow probabilities are used to recover the executed path and the corresponding input values triggering the assertion.

3 Evaluation

We evaluate DASA in SV-COMP 2026 [2] purely as a falsifier for the Java *valid-assert* track, focusing on finding assertion-triggering inputs and producing violation witnesses, not on proving correctness. To ensure a fair comparison, we report results for DASA and competing tools only within this scope.

Due to an issue in witness creation, 55 detected violations could not be validated. We therefore report the raw number of correctly solved violation tasks. In its first participation, DASA solved 293 tasks, placing it behind MLB (527) [3], JBMC (512) [5], GDart (444) [14], JDART (439) [13], SWAT (442) [10] and javaranger (294) [6], but ahead of spf (263) [20], coastal (239) [25], JayHorn (160) [23] and JLisa (39) [1]. Notably, DASA achieved 22 unique solves, i.e., verification tasks that no other tool could solve. This is tied for the highest count (together with MLB), slightly ahead of JBMC (21), and substantially exceeding GDart (3), JDart (2), SWAT (1), and the remaining tools (0).

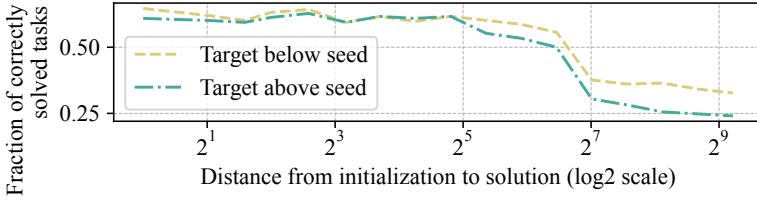


Fig. 2. Success rate vs. the distance between the initial seed and the target input. A success rate of 100% corresponds to the overall score achieved by DASA in SV-COMP.

Limitations. As an early-stage research prototype, DASA exhibits several limitations and open challenges inherent to gradient-based program analysis:

First, tasks that require very high floating-point precision can be challenging: since DASA relies on gradient-based optimization, it may converge to values that are close but not exact (e.g., producing 99.9999 instead of 100.0).

Second, the effective optimization region is limited: if a valid solution lies far from the initial seed, the optimizer may fail to reach it. This sensitivity is illustrated in Figure 2. A substantial fraction of benchmarks are only solved due to the initial seed choice, as reflected by an immediate drop to a 62% success rate when no constant values are drawn from the target. Moreover, gradient-based optimization typically bridges distances only up to approximately 2^6 , beyond which success rates decline sharply. While optimization towards smaller target values appears easier than towards larger ones, this effect is influenced by a benchmark bias, as “greater-than” comparisons occur substantially more frequently.

Finally, certain program semantics are inherently difficult to express in a smooth differentiable form, e.g., discrete operations such as integer bit flips.

However, overall, the evaluation demonstrates that fully differentiable program analysis is not only feasible but already competitive in practice, despite DASA being an early-stage and highly experimental prototype. While the approach currently exhibits limitations, differentiable analysis might constitute a promising and complementary direction for program analysis, with an important next step being its extension to support correctness verification, for example by detecting cases in which the loss cannot be fully minimized.

4 Software Project

DASA is developed at the Institute for IT Security at the University of Lübeck and is publicly available on GitHub [16] under the MIT license. The repository includes installation instructions, documentation, and usage examples [15].

For the competition, DASA in version 0.1 is evaluated in the Java *valid-assert* category under a fixed global configuration. Optimization is performed using the Adam optimizer [8] for 2000 steps, with a learning rate of 0.1 and up to 500 restart iterations per target. To balance exploration and convergence, sigmoid-based annealing is applied with parameters ranging from 0.001 to 1.0.

5 Data-Availability Statement

The version of DASA used for the SV-COMP 2026 Java *valid-assert* category is available at Zenodo [18] and on GitHub [17].

6 Acknowledgments

This work has been supported by the *Federal Ministry of Research, Technology and Space of Germany (BMFTR)* through the *SILGENTAS* project and by the *State of Schleswig-Holstein* through the *VeDS* project.

References

1. V. Arceri, L. Negrini, G. Zanatta, F. Bianchi, T. Lisovenko, L. Olivieri, and P. Ferrara. JLiSA: The Java frontend of the library for static analysis (competition contribution). In *Proc. TACAS (2)*, LNCS 16506. Springer, 2026.
2. D. Beyer and J. Strejček. Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In *Proc. TACAS (2)*, LNCS 16506. Springer, 2026.
3. Lei Bu, Yongjuan Liang, Zhunyi Xie, Hong Qian, Yi-Qi Hu, Yang Yu, Xin Chen, and Xuandong Li. Machine learning steered symbolic execution framework for complex software code. *Formal Aspects Comput.*, 33(3):301–323, 2021.
4. Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 711–725. IEEE Computer Society, 2018.
5. L. C. Cordeiro, D. Kröning, and P. Schrammel. JBMC: Bounded model checking for Java bytecode (competition contribution). In *Proc. TACAS (3)*, LNCS 11429, pages 219–223. Springer, 2019.
6. S. Hussein, Q. Yan, S. McCamant, V. Sharma, and M. Whalen. JAVA RANGER: Supporting string and array operations (competition contribution). In *Proc. TACAS (2)*, LNCS 13994, pages 553–558. Springer, 2023.
7. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.
8. Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.
9. Minghao Liu, Kunhang Lv, Pei Huang, Rui Han, Fuqi Jia, Yu Zhang, Feifei Ma, and Jian Zhang. Nrago: Solving SMT(NRA) formulas with gradient-based optimization. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 2046–2049. IEEE, 2023.
10. N. Loose, F. Sieck, F. Mächtle, and T. Eisenbarth. SWAT: Improvements to the symbolic executor (competition contribution). In *Proc. TACAS (2)*, LNCS 16506. Springer, 2026.
11. Nils Loose, Felix Mächtle, Florian Sieck, and Thomas Eisenbarth. SWAT: modular dynamic symbolic execution for java applications using dynamic instrumentation (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of*

- the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III*, volume 14572 of *Lecture Notes in Computer Science*, pages 399–405. Springer, 2024.
12. Felix Mächtle, Nils Loose, Jan-Niclas Serr, Jonas Sander, and Thomas Eisenbarth. Autostub: Genetic programming-based stub creation for symbolic execution. In *IEEE/ACM International Workshop on Search-Based and Fuzz Testing, SBFT 2025, Ottawa, ON, Canada, April 28-29, 2025*, pages 19–22. IEEE, 2025.
 13. M. Mues and F. Howar. JDART: Portfolio solving, breadth-first search and SMT-Lib strings (competition contribution). In *Proc. TACAS (2)*, LNCS 12652, pages 448–452. Springer, 2021.
 14. M. Mues and F. Howar. GDART (competition contribution). In *Proc. TACAS (2)*, LNCS 13244, pages 435–439. Springer, 2022.
 15. Felix Mächtle, Jan-Niclas Serr, Nils Loose, and Thomas Eisenbarth. DASA Documentation. <https://uzl-its.github.io/dasa-program-analysis/>. Accessed 12/2025.
 16. Felix Mächtle, Jan-Niclas Serr, Nils Loose, and Thomas Eisenbarth. DASA Main Repository. <https://github.com/UzL-ITS/dasa-program-analysis>. Accessed 12/2025.
 17. Felix Mächtle, Jan-Niclas Serr, Nils Loose, and Thomas Eisenbarth. DASA, Version 0.1 Release. <https://github.com/UzL-ITS/dasa-program-analysis/releases/tag/v0.1>. Accessed 12/2025.
 18. Felix Mächtle, Jan-Niclas Serr, Nils Loose, and Thomas Eisenbarth. DASA. <https://doi.org/10.5281/zenodo.17693012>, November 2025.
 19. Maria-Irina Nicolae, Max Eisele, and Andreas Zeller. Revisiting neural program smoothing for fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 133–145. ACM, 2023.
 20. Y. Noller, C. S. Păsăreanu, X.-B. D. Le, W. Visser, and A. Fromherz. Symbolic PATHFINDER for SV-COMP (competition contribution). In *Proc. TACAS (3)*, LNCS 11429, pages 239–243. Springer, 2019.
 21. Oracle Corporation. GraalVM. <https://www.graalvm.org/>, 2025. Accessed: 12/2025.
 22. Adam Paszke, Sam Gross, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.
 23. A. Shamakhi, H. Hojjat, and P. Rümmer. Towards string support in JAYHORN (competition contribution). In *Proc. TACAS (2)*, LNCS 12652, pages 443–447. Springer, 2021.
 24. Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 803–817. IEEE, 2019.
 25. W. Visser and J. Geldenhuys. COASTAL: Combining concolic and fuzzing for Java (competition contribution). In *Proc. TACAS (2)*, LNCS 12079, pages 373–377. Springer, 2020.
 26. Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. 1 2019.

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





EmergenTheta: Experimental Analyses within the Theta Framework (Competition Contribution)

Milán Mondok[✉], Csanád Telbisz^{*}, Levente Bajczi[✉],
Dániel Kovács[✉], Mihály Dobos-Kovács[✉], and Vince Molnár[✉]

Department of Artificial Intelligence and Systems Engineering
Budapest University of Technology and Economics, Budapest, Hungary
`mondok@mit.bme.hu`

Abstract. EMERGENTHETA is our sandbox for emerging analyses within the THETA framework. Previously established analyses have been migrated to THETA, and EMERGENTHETA focuses exclusively on novelties for both counterexample-guided abstraction refinement over control-flow automata and reachability analyses over symbolic transition systems. The 2026 submission presents a combined predicate-explicit abstraction domain, multithreaded symbolic transition system encodings, on-the-fly reachability and counterexample generation for generalized saturation, and an IC3 backend. This is the first year that we were able to submit EMERGENTHETA to all categories, representing a significant step in the evolution of THETA, and the competition results indicate the practical viability of the new techniques.

1 Verification Approach

The algorithms of EMERGENTHETA fall into two categories: 1) the *counterexample-guided abstraction refinement* (CEGAR) algorithm that conducts explicit step-by-step exploration of the abstract state space by building an abstract reachability graph as described in [11], which operates directly on the control-flow automaton (CFA) description created from the C program, and 2) various reachability algorithms that use the low-level symbolic transition system (STS) description created from the CFA. The latter category contains the *generalized saturation* (GSAT) [12,14], the *property-directed reachability* (IC3/PDR) [10] and *bounded model checking* (BMC) [7] algorithms. See Figure 1 for an overview, presenting novelties exclusive to EMERGENTHETA (not used by the mainline THETA).

Novelties related to the CEGAR algorithm. The *combined abstraction domain* [13] is a combination of the predicate and explicit-value abstraction domains, which tracks a subset of the variables explicitly, while also tracking the truth-values of a set of Boolean predicates. States are pairs of explicit-value and

* Jury member representing EMERGENTHETA at SV-COMP 2026.

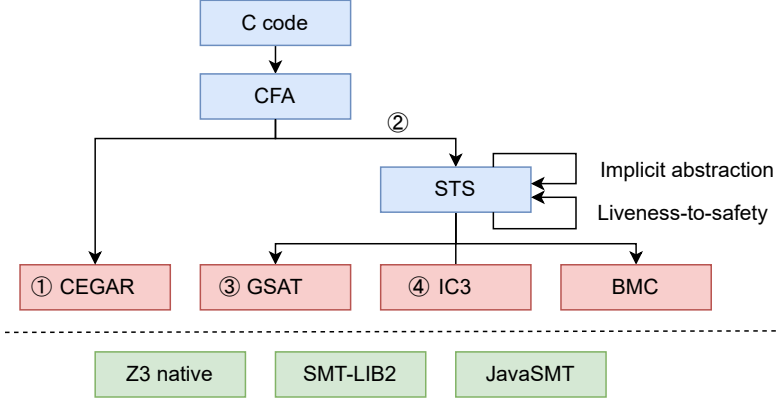


Fig. 1: Overview of the verification approach of EMERGENTHETA. 2026 additions are ① *combined domain* for CEGAR, ② *multithreaded CFA-to-STS transformation*, ③ *on-the-fly reachability and counterexample generation* for the generalized saturation algorithm, ④ an *IC3* algorithm backend. Blue, red and green rectangles respectively denote *task representations*, *algorithm backends* and *SMT-solver backends*.

predicate states, the transfer function reasons about both domains simultaneously. By default all variables are tracked through predicates and the algorithm uses heuristics to decide if a variable should be tracked explicitly instead.

A preprocessing step collects all literal expressions co-occurring with each variable (in the CFA, this means assumptions and assignments involving both the variable and the literal). During refinement, if a new predicate introduces previously unseen literals with the variable, we switch to explicit tracking. This helps detect “predicate counting”, which typically occurs when the solver returns interpolants containing literals absent from the model.

Novelties related to STS-based algorithms. This class of algorithms operates on *symbolic transition systems* (STS), which encode reachability problems using three SMT formulae (the initial states, the transition relation, and the safety invariant). Before verification, we translate the CFA representation into an STS, encoding control locations as variables. As a recent extension, this transformation supports *multithreaded programs* by statically determining an upper bound on the number of threads and encoding each thread’s control location with a separate variable. Programs that may spawn an unbounded number of threads cannot be encoded finitely; while array-based SMT encodings are possible, we found them inefficient. In such cases, we encode only a finite number of threads and report unknown if no bug is found and threads may be created unboundedly. This novel transformation is only used by EMERGENTHETA and not the mainline THETA.

Generalized saturation [12] utilizes substitution diagrams [14] to dynamically create multi-valued decision diagrams from the SMT descriptions of the model, enabling the use of the symbolic algorithm that was originally defined on Petri nets. However, saturation performs enumeration of the entire state space and cannot return counterexamples. A novel addition to the algorithm this year is *on-the-fly reachability*, which we conduct using a modification of constrained saturation that traverses the set of target states simultaneously with the transition relation and stops enumeration whenever a target state is reached. A further addition is *counterexample generation*: we use depth-first search to find a counterexample in the intersection of 1) states from which the target states can be reached, and 2) the states explored during on-the-fly reachability. Furthermore, the multithreaded CFA-to-STS transformation allowed us to use generalized saturation on multithreaded C programs this year, where we expect it to perform well, as it is designed to exploit symmetries in asynchronous behaviour.

A new addition to the set of algorithms this year is *property directed reachability* (IC3) [10], which analyzes the model incrementally through the proof of several lemmas, eventually forming an inductive invariant proving the system correct or directing the search towards a counterexample.

2 Software Architecture

EMERGENTHETA is a JVM-based tool, implemented in Kotlin, with legacy components written in Java. EMERGENTHETA is part of the THETA framework [11], and can therefore use its integrated frontend- and solver infrastructure. Parsing C files is done using a custom ANTLR grammar [2]. SMT solvers can be used in three ways: Z3 [16] natively via its Java API, other solvers (e.g., MathSAT5 [8], CVC5 [3]) via SMT-LIBv2 [9], and JavaSMT [1]. For SV-COMP’26, we use Z3 4.5.0 and 4.14.0; MathSAT5 5.6.12; and CVC5 1.2.0.

For 2026, we completely reworked the transformation engine that manages model transformations to handle the wide number of input formalisms, algorithm backends and model transformations flexibly using a pipeline architecture with active passes. We provide support for *overflow*, *memory safety*, and *data race* properties via transformations that modify the control-flow automata. We also provide support for *termination* properties through a *liveness-to-safety* [6] transformation, implemented as an STS-to-STS transformation. These transformations represent a significant step in the architecture of THETA, but are not unique to EMERGENTHETA.

For SV-COMP, we combine multiple configurations in sequential portfolios and select the appropriate portfolio based on traits of the C program (e.g., presence of arrays, multithreading, floating point operations). Each configuration is executed with a predefined timeout and we switch to a fallback SMT solver in case of a solver error. For multithreaded programs, the portfolio applies GSAT, BMC, and combined CEGAR in this order. For single-threaded programs, the portfolio uses combined CEGAR using multiple refinement strategies, followed by GSAT and IC3, both augmented with implicit predicate abstraction [17]. For

Configuration	no-data-race		no-overflow		termination		unreach-call		memsafety	
	safe	unsafe	safe	unsafe	safe	unsafe	safe	unsafe	safe	unsafe
GSAT	557	32	163	0	0	0	90	124	191	0
Combined CEGAR	5	10	392	214	0	0	1136	364	204	12
IC3	0	0	0	3	0	0	0	2	0	0
BMC	7	9	8	3	0	0	10	16	16	0
THETA fallback	1	2	191	15	0	0	674	122	15	2
Liveness-CEGAR	0	0	0	0	268	113	0	0	0	0
Liveness-to-safety	0	0	0	0	16	11	0	0	0	0
$\sum$ Correct	570	53	754	235	284	124	1910	628	426	14
$\sum$ Wrong	0	1	1	12	0	0	5	2	2	13
$\sum$ Unconfirmed correct	0	14	0	33	0	0	118	23	0	6

Table 1: Verification results per configuration and property

termination tasks, the portfolio begins with liveness-CEGAR [15], followed by BMC and k-induction using the liveness-to-safety [6] transformation. Portfolios fall back to the stable portfolio (`--portfolio STABLE`) if all configurations fail.

3 Strengths and Weaknesses of the Approach

Table 1 presents our results [4] per property (*memsafety* combines all properties related to memory safety). The combined CEGAR and the GSAT algorithms managed to solve the most tasks partly due to the fact that they are placed at the start of the portfolios (explaining the relatively low number of solved tasks by the IMC and BMC algorithms, which were placed further behind in the portfolios). GSAT performed very promisingly on multithreaded problems, and was able to return counterexamples for the *unreach-call* property. Using liveness-CEGAR, we were able to solve more *termination* tasks this year, than last year using the transformation of TransVer [5] (408 vs 217). EMERGENTHETA produced a few wrong results (36 in total), the main reason behind which are parsing issues in the frontend of our tool. Overall, EMERGENTHETA solved 4998 tasks this year, achieving an overall score of 11248.

4 Software Project, Tool Setup and Configuration

EMERGENTHETA is integrated into the THETA verification framework maintained by the Critical Systems Research Group¹ of the Budapest University of Technology and Economics. Using the competition archive [18] for software verification, EMERGENTHETA can be started using the following input options: `./theta-start.sh <input> --svcomp --portfolio EMERGENT`. This command starts the EMERGENT portfolio that we used at SV-COMP 2026. This portfolio handles all categories of C verification at SV-COMP using our emerging configuration options. For the first time this year, we participate in all categories.

¹ <https://ftsrg.mit.bme.hu/en/>

5 Data-Availability Statement

The project is available open-source on GitHub² under an Apache 2.0 license, and the binary release of the competition version (6.27.9) is published on Zenodo [18].

Funding. This research was partially funded by the EKÖP-25-3 New National Excellence Program under project numbers EKÖP-25-3-BME-243, and EKÖP-25-3-BME-22, and the Doctoral Excellence Fellowship Programme under project numbers 400434/2023 and 400443/2023; funded by the NRDI Fund of Hungary.

References

1. Baier, D., Beyer, D., Friedberger, K.: JavaSMT 3: Interacting with SMT Solvers in Java. In: Silva, A., Leino, K.R.M. (eds.) *Computer Aided Verification. Lecture Notes in Computer Science*, vol. 12760, pp. 195–208. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_9
2. Bajcsi, L., Ádám, Z., Molnár, V.: C for Yourself: Comparison of Front-End Techniques for Formal Verification. In: *10th FormaliSEICSE 2022*. pp. 1–11. ACM (2022). <https://doi.org/10.1145/3524482.3527646>
3. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: Fisman, D., Rosu, G. (eds.) *TACAS 2022*. pp. 415–442. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
4. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In: *Proc. TACAS (2)*. LNCS 16506, Springer (2026)
5. Beyer, D., Jankola, M., Lingsch-Rosenfeld, M., Xia, T., Zheng, X.: A modular program-transformation framework for reducing specifications to reachability (2025), <https://arxiv.org/abs/2501.16310>
6. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. In: *FMICS 2002, ICALP 2002 Satellite Workshop (2002)*. [https://doi.org/10.1016/S1571-0661\(04\)80410-9](https://doi.org/10.1016/S1571-0661(04)80410-9)
7. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: *TACAS (1999)*. https://doi.org/10.1007/3-540-49059-0_14
8. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: *TACAS 2013, LNCS*, vol. 7795, pp. 93–107. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
9. Dobos-Kovács, M., Vörös, A.: Evaluation of SMT solvers in abstraction-based software model checking. In: *Proceedings of the 11th Latin-American Symposium on Dependable Computing*. p. 109–116. LADC '22, Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3569902.3570187>
10. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. p. 125–134. FMCAD '11, FMCAD Inc, Austin, Texas (2011), <https://dl.acm.org/doi/10.5555/2157654.2157675>

² <https://github.com/ftsrg/theta/releases/tag/svcomp26>

11. Hajdu, Á., Micskei, Z.: Efficient Strategies for CEGAR-based Model Checking. *Journal of Automated Reasoning* **64**(6), 1051–1091 (2020). <https://doi.org/10.1007/s10817-019-09535-x>
12. Molnár, V., Majzik, I.: Saturation Enhanced with Conditional Locality: Application to Petri Nets. In: Donatelli, S., Haar, S. (eds.) *Application and Theory of Petri Nets and Concurrency*. pp. 342–361. Springer International Publishing, Cham (2019)
13. Mondok, M.: Efficient formal verification of component-based engineering models. Master’s thesis, Budapest University of Technology and Economics (2022)
14. Mondok, M., Molnár, V.: Efficient Manipulation of Logical Formulas as Decision Diagrams. In: Renczes, B. (ed.) *Proceedings of the 31st PhD Mini-Symposium*. pp. 61–65. Budapest University of Technology and Economics, Department of Measurement and Information Systems (2024). <https://doi.org/10.3311/MINISY2024-012>
15. Mondok, M., Vörös, A.: Abstraction-based model checking of linear temporal properties. In: Renczes, B. (ed.) *Proceedings of the 27th PhD Mini-Symposium*. pp. 29–32. Budapest University of Technology and Economics, Department of Measurement and Information Systems (2020)
16. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: *TACAS 2008, LNCS*, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
17. Tonetta, S.: Abstract Model Checking without Computing the Abstraction. In: Cavalcanti, A., Dams, D.R. (eds.) *FM 2009: Formal Methods*. pp. 89–105. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
18. Ádám, Z., Bajcsi, L., Dobos-Kovács, M., Molnár, V., Mondok, M., Somorjai, M., Szekeres, D., Telbisz, C., Kovács, D., Rippl, B.R.: Emergentheta - verifier archive (Dec 2025). <https://doi.org/10.5281/zenodo.17779912>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





GOBLINT: A Portfolio for Mixed Flow-Sensitive Abstract Interpretation (Competition Contribution)

Simmo Saan^{1(✉)*}, Ali Rasim Kocal², Michael Petter²,
Karoliine Holter¹, Julian Erhard^{2,3}, Michael Schwarz⁴,
Vesal Vojdani¹, and Helmut Seidl²

¹ University of Tartu, Tartu, Estonia

`simmo.saan@ut.ee`

² Technische Universität München, Garching, Germany

³ Ludwig-Maximilians-Universität München, Munich, Germany

⁴ National University of Singapore, Singapore

Abstract. GOBLINT is a static analyzer for C programs based on abstract interpretation. It provides a flexible framework for combining analyses and specifying what information is tracked flow- and context-sensitively. For SV-COMP 2026, we adopt a sequential portfolio of configurations of increasing precision that exploits GOBLINT’s rich family of abstract domains and sensitivity options. We have improved the analyses by adding bit-level and sparse affine equalities domains, new forms of path-sensitivity, and a context gas mechanism that controls context-sensitivity for recursive programs. GOBLINT continues its unblemished track record of sound verification across all SV-COMP benchmarks.

1 Verification Approach

GOBLINT is a highly-configurable abstract interpreter for C programs. In particular, it provides a thread-modular and scalable analysis of concurrent programs [21, 26, 27]. The various analyses implemented in GOBLINT define a side-effecting constraint system [2] to be solved. Constraint systems of this kind are ideal for expressing *mixed flow-sensitivity* [29], where parts of the analysis state are tracked in a flow-sensitive and others in a flow-insensitive manner. The rest of this section describes advances since the previous tool papers [18, 21, 22], while the following section presents the new sequential portfolio.

Domains. Analysis of bit-level code has been improved by making the bitwise operators of existing integer domains (e.g., intervals) more precise and by adding a new non-relational bitfield domain [13]. We also improved relational integer domains. With our array-based implementation of affine equalities [11], we observed that only 15% of coefficients are non-zero in practice. Switching to a sparse representation using functional lists yielded speedups between a factor of 2 and 10.

* SV-COMP jury member

Analyses. Two new sources of path-sensitivity have been introduced. First, sensitivity w.r.t. the set of branches taken implements a form of trace partitioning [9, 17]. Second, sensitivity w.r.t. the abstract values of the non-relational analysis yields a variant of explicit-state model checking [4]. Activated for multi-threaded programs, they are instances of ego-lane-derived digests [23, 27] and abstract digests [24], respectively. Abstract calling contexts can now be controlled using context gas [7, 8]. Called functions are analyzed with decremented gas or context-insensitively if the gas runs out. Limiting the number of contexts aids analysis termination and improves results in particular on recursive programs.

Solvers. The top-down constraint solver [28] has gained three more improvements. First, it can now delay widening for both flow-sensitive and flow-insensitive unknowns. The latter is particularly important for precise analysis of concurrent programs. Second, values of such unknowns can be narrowed to further increase precision [29]. Third, the constraint systems for concurrent programs use weak dependencies which reduce unnecessary recomputation by the solver [12].

Witnesses. In the *C.Concurrency* category, GOBLINT now generates YAML witnesses using ghost instrumentation [6]. The thread-modular invariants computed by GOBLINT are conveniently expressed using ghost variables corresponding to the locking state of mutexes. Furthermore, GOBLINT now avoids outputting trivial and redundant witness invariants which can significantly increase witness size and validation effort. For example, instead of directly exporting all octagon constraints as witness invariants, GOBLINT skips those which can be inferred from interval constraints by the closure algorithm. The overall quality of witnesses has been improved by fixing many issues concerning variables in invariant expressions (e.g., regarding scope) and locations of invariants (e.g., related to complex loops and initializers).

2 Software Architecture

GOBLINT is implemented in OCAML and uses a maintained fork⁵ of CIL [16] as its parser frontend for the C language. It relies on GCC for preprocessing C code and APRON [10] for some relational analyses. No other major libraries or external tools are required.

GOBLINT has a modular architecture [1] which allows a combination of analyses to be selected and automatically configured at runtime [22]. Analysis definitions specify their abstract domains and transfer functions, which can communicate with other analyses using predefined queries and events. The combination of active analyses together with the control-flow graphs of the functions yields a side-effecting constraint system, which is solved using a local generic solver [28]. Its solution is post-processed to determine the verdict and construct a witness.

⁵ <https://github.com/goblint/cil>

Sequential Portfolio. A major change for SV-COMP 2026 is the introduction of a sequential portfolio, which has been a successful approach for other tools [3, 14, 15, 31]. To this end, a PYTHON script executes GOBLINT with various configurations in sequence until a definite verdict is obtained, the resources are exhausted or all configurations have been tried.

We have handcrafted six configurations (called *levels*) of increasing precision and resource consumption:

Level 0 combines constant propagation and exclusion sets with context gas of 30, based on the evaluation of Erhard et al. [7, 8].

Level 1 additionally uses intervals (integer and floating-point) and symbolic variable equalities, while allowing the autotuner [22] to enable further domains (congruences, enumerator sets, octagons). This is closest to the single configuration used by GOBLINT in SV-COMP 2025 and before.

Level 2 unconditionally activates various integer domains, both non-relational (congruences, enumerator sets, bitfields) and relational (octagons, affine equalities, structs). It also replaces context gas with equal widening delay.

Level 3 introduces path-sensitivity based on the set of branches taken.

Level 4 switches to path-sensitivity w.r.t. the non-relational value domain and reverts to relational analyses being conditionally enabled by the autotuner.

Level 5 unconditionally activates polyhedra, doubles the widening delay and enables the narrowing of flow-insensitive unknowns.

This order of configurations was, however, not chosen manually. First, we separately evaluated each configuration on the SV-COMP 2025 benchmark suite. This yielded the verdict and runtime of each configuration on each benchmark. Using this information, we could quickly simulate the results of a hypothetical sequential portfolio consisting of these configurations in any order. By repeating the simulation for every permutation, we arrived at the above order which maximizes the weighted score for the *C.Overall* category. Having time limits for individual levels was not beneficial with these configurations.

Compared to our previous single configuration (level 1), this portfolio achieves 12% more correct results. It solves 99.1% of tasks solvable by the virtual best of these configurations, an oracle that chooses the best configuration for each task.

3 Strengths and Weaknesses

GOBLINT continued its soundness streak in SV-COMP 2025 and 2026: it did not produce any false negatives across all SV-COMP benchmarks. However, due to over-approximation, it cannot report definite violations without compromising its correctness. GOBLINT also maintained its lead in the *C.no-data-race.Concurrency* category in SV-COMP 2025 and 2026. A recent paper [25] provides semantic foundations for our data race analysis [30] and attributes its success to the combination of efficient reasoning on locksets refined with just enough may-happen-in-parallel information derived from thread creation histories.

Table 1. Cumulative and proportional distribution of results across portfolio levels.

Level	<i>C.Overall</i> score		Solved tasks		Timeouts/out of memories		
0	19,258	84.9%	8,765	84.6%	511		3.0%
1	21,380 (+2,122)	9.4%	9,965 (+1,200)	11.6%	1,072	(+561)	3.3%
2	21,609 (+229)	1.0%	10,077 (+112)	1.1%	3,108	(+2,036)	12.0%
3	22,158 (+549)	2.4%	10,233 (+156)	1.5%	13,349	(+10,241)	60.3%
4	22,670 (+512)	2.3%	10,362 (+129)	1.2%	15,631	(+2,282)	13.4%
5	22,673 (+3)	0.0%	10,364 (+2)	0.0%	16,971	(+1,340)	7.9%

Table 1 shows how each level in the portfolio contributes to the overall results in SV-COMP 2026 (with witness validation). Despite its simplicity, level 0 already achieves over 84% of the score, with level 1 contributing an additional 9%. While levels 3 and 4 contribute only 2.7% of solved tasks, they amount to 4.7% of the score. Meanwhile, the path-sensitivity at these levels is the reason for most timeouts overall. Future work may diversify the configurations to reduce their overlap and solve more tasks overall, or parallelize parts of the portfolio.

4 Tool Setup and Configuration

GOBLINT version `svcomp26-0-gbb55b70d9` [20] participated in *all* categories of SV-COMP 2026 [5], except *C.FalseOverall*. Both the tool-info module and the benchmark definition for SV-COMP are named `goblnt`. They correspond to running the tool as follows:

```
./goblnt_runner.py --portfolio-conf conf/svcomp26/seq.txt \
                  --set ana.specification property.prp \
                  --set exp.architecture 64bit input.c
```

This runs GOBLINT itself, e.g., in the first configuration, as follows:

```
./goblnt --conf conf/svcomp26/common.json \
         --conf conf/svcomp26/verify.json \
         --conf conf/svcomp26/level00.json \
         --set ana.specification property.prp \
         --set exp.architecture 64bit input.c
```

The configuration is composed of multiple files to reduce duplication between levels and with GOBLINT VALIDATOR [19], which now uses the same portfolio.

5 Software Project and Contributors

GOBLINT development takes place on GitHub, while related publications are listed on its website.⁶ It is an MIT-licensed project initiated by Technische Universität München and the University of Tartu, with additional contributors at Ludwig-Maximilians-Universität München and National University of Singapore.

⁶ <https://github.com/goblnt/analyzer> and <https://goblnt.in.tum.de>

Acknowledgments. This work was supported by Deutsche Forschungsgemeinschaft (DFG) – 378803395/2428 CONVEY 2, the European Union and the [Estonian Research Council](#) via projects PRG2764 and TEM-TA119. This work has benefitted from the participation of Helmut Seidl, Michael Schwarz and the SV-COMP organizer Dirk Beyer in the Dagstuhl Seminar 25421 “Sound Static Program Analysis in Modern Software Engineering”. We would like to thank everyone who has contributed to GOBLINT over the years, especially the students who contributed the new features described in Section 1.

Data Availability Statement. All data of SV-COMP 2026 are archived as described in the competition report [5] and available on the [competition website](#). This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of GOBLINT as used in the competition is archived on Zenodo [20], while the corresponding source code is available on [GitHub](#).

Bibliography

- [1] Apinis, K.: Frameworks for analyzing multi-threaded C. Ph.D. thesis, Technische Universität München (2014)
- [2] Apinis, K., Seidl, H., Vojdani, V.: Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In: APLAS ’12, pp. 157–172, Springer (2012), DOI: [10.1007/978-3-642-35182-2_12](#)
- [3] Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPACHECKER 2.3 with strategy selection. In: TACAS ’24, pp. 359–364, Springer (2024), DOI: [10.1007/978-3-031-57256-2_21](#)
- [4] Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: FASE ’13, pp. 146–162, Springer (2013), DOI: [10.1007/978-3-642-37057-1_11](#)
- [5] Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In: TACAS ’26, Springer (2026)
- [6] Erhard, J., Bentele, M., Heizmann, M., Klumpp, D., Saan, S., Schüssele, F., Schwarz, M., Seidl, H., Tilscher, S., Vojdani, V.: Correctness witnesses for concurrent programs: Bridging the semantic divide with ghosts. In: VMCAI ’25, pp. 74–100, Springer (2025), DOI: [10.1007/978-3-031-82700-6_4](#)
- [7] Erhard, J., Schinabeck, J.F., Schwarz, M., Seidl, H.: When to stop going down the rabbit hole: Taming context-sensitivity on the fly. In: SOAP ’24, pp. 35–44, ACM (2024), DOI: [10.1145/3652588.3663321](#)
- [8] Erhard, J., Schinabeck, J.F., Schwarz, M., Seidl, H.: Context gas and friends: taming context-sensitivity on the fly. Int. J. Softw. Tools Technol. Transf. **27**(2), 289–307 (2025), DOI: [10.1007/S10009-025-00803-3](#)
- [9] Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: SAS ’98, pp. 200–214, Springer (1998), DOI: [10.1007/3-540-49727-7_12](#)
- [10] Jeannet, B., Miné, A.: APRON: A library of numerical abstract domains for static analysis. In: CAV ’09, pp. 661–667, Springer (2009), DOI: [10.1007/978-3-642-02658-4_52](#)

- [11] Karr, M.: Affine relationships among variables of a program. *Acta Informatica* **6**(2), 133–151 (1976), DOI: [10.1007/BF00268497](https://doi.org/10.1007/BF00268497)
- [12] Kocal, A.R., Schwarz, M., Saan, S., Seidl, H.: Same engine, multiple gears: Parallelizing fixpoint iteration at different granularities. In: *TACAS '26*, Springer (2026)
- [13] Miné, A.: Abstract domains for bit-level machine integer and floating-point operations. In: *WInG '12*, pp. 55–70, EasyChair (2013), DOI: [10.29007/b63g](https://doi.org/10.29007/b63g)
- [14] Monat, R., Ouadjaout, A., Miné, A.: MOPSA-C: Modular domains and relational abstract interpretation for C programs. In: *TACAS '23*, pp. 565–570, Springer (2023), DOI: [10.1007/978-3-031-30820-8_37](https://doi.org/10.1007/978-3-031-30820-8_37)
- [15] Monat, R., Ouadjaout, A., Miné, A.: MOPSA-C with trace partitioning and autosuggestions. In: *TACAS '25*, pp. 229–235, Springer (2025), DOI: [10.1007/978-3-031-90660-2_17](https://doi.org/10.1007/978-3-031-90660-2_17)
- [16] Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: *CC '02*, pp. 213–228, Springer (2002), DOI: [10.1007/3-540-45937-5_16](https://doi.org/10.1007/3-540-45937-5_16)
- [17] Rival, X., Mauborgne, L.: The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.* **29**(5), 26–es (2007), DOI: [10.1145/1275497.1275501](https://doi.org/10.1145/1275497.1275501)
- [18] Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: GOBLINT: Abstract interpretation for memory safety and termination. In: *TACAS '24*, pp. 381–386, Springer (2024), DOI: [10.1007/978-3-031-57256-2_25](https://doi.org/10.1007/978-3-031-57256-2_25)
- [19] Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: GOBLINT VALIDATOR: Correctness witness validation by abstract interpretation. In: *TACAS '24*, pp. 335–340, Springer (2024), DOI: [10.1007/978-3-031-57256-2_17](https://doi.org/10.1007/978-3-031-57256-2_17)
- [20] Saan, S., Kocal, A.R., Petter, M., Holter, K., Erhard, J., Schwarz, M., Vojdani, V., Seidl, H.: GOBLINT in SV-COMP 2026 (2025), DOI: [10.5281/zenodo.17642247](https://doi.org/10.5281/zenodo.17642247), tool artifact
- [21] Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: GOBLINT: Thread-modular abstract interpretation using side-effecting constraints. In: *TACAS '21*, pp. 438–442, Springer (2021), DOI: [10.1007/978-3-030-72013-1_28](https://doi.org/10.1007/978-3-030-72013-1_28)
- [22] Saan, S., Schwarz, M., Erhard, J., Pietsch, M., Seidl, H., Tilscher, S., Vojdani, V.: GOBLINT: Autotuning thread-modular abstract interpretation. In: *TACAS '23*, vol. 2, pp. 547–552, Springer (2023), DOI: [10.1007/978-3-031-30820-8_34](https://doi.org/10.1007/978-3-031-30820-8_34)
- [23] Schwarz, M.: Thread-Modular Abstract Interpretation: The Local Perspective. Ph.D. thesis, Technische Universität München (2025), URL: <https://d-nb.info/137113345X/34>
- [24] Schwarz, M., Erhard, J.: The digest framework: concurrency-sensitivity for abstract interpretation. *Int. J. Softw. Tools Technol. Transf.* **26**(6), 727–746 (2024), DOI: [10.1007/s10009-024-00773-y](https://doi.org/10.1007/s10009-024-00773-y)
- [25] Schwarz, M., Erhard, J.: Data race detection by digest-driven abstract interpretation. In: *VMCAI '26*, Springer (2026), DOI: [10.1007/978-3-032-15700-3_15](https://doi.org/10.1007/978-3-032-15700-3_15)

- [26] Schwarz, M., Saan, S., Seidl, H., Apinis, K., Erhard, J., Vojdani, V.: Improving thread-modular abstract interpretation. In: SAS '21, pp. 359–383, Springer (2021), DOI: [10.1007/978-3-030-88806-0_18](https://doi.org/10.1007/978-3-030-88806-0_18)
- [27] Schwarz, M., Saan, S., Seidl, H., Erhard, J., Vojdani, V.: Clustered relational thread-modular abstract interpretation with local traces. In: ESOP '23, pp. 28–58, Springer (2023), DOI: [10.1007/978-3-031-30044-8_2](https://doi.org/10.1007/978-3-031-30044-8_2)
- [28] Seidl, H., Vogler, R.: Three improvements to the top-down solver. *Math. Struct. Comput. Sci.* **31**(9), 1090–1134 (2021), DOI: [10.1017/S0960129521000499](https://doi.org/10.1017/S0960129521000499)
- [29] Stemmler, F., Schwarz, M., Erhard, J., Tilscher, S., Seidl, H.: Taking out the toxic trash: Recovering precision in mixed flow-sensitive static analyses. In: PLDI '25, pp. 1167–1193, ACM (2025), DOI: [10.1145/3729297](https://doi.org/10.1145/3729297)
- [30] Vojdani, V., Apinis, K., Rötov, V., Seidl, H., Vene, V., Vogler, R.: Static Race Detection for Device Drivers: The Goblin Approach. In: ASE '16, pp. 391–402, ACM (2016), DOI: [10.1145/2970276.2970337](https://doi.org/10.1145/2970276.2970337)
- [31] Wendler, P.: CPACHECKER with sequential combination of explicit-state analysis and predicate analysis. In: TACAS '13, pp. 613–615, Springer (2013), DOI: [10.1007/978-3-642-36742-7_45](https://doi.org/10.1007/978-3-642-36742-7_45)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





GOBLITCH: Combining Abstract Interpretation with Symbolic Execution via Witnesses (Competition Contribution)

Karoliine Holter¹✉, Paulína Ayaziová², Simmo Saan¹,
Jan Strejček², and Vesal Vojdani¹

¹ University of Tartu, Tartu, Estonia

{karoliine.holter, simmo.saan, vesal.vojdanil}@ut.ee

² Masaryk University, Brno, Czech Republic

{xayaziov, strejcek}@fi.muni.cz

Abstract. GOBLITCH is a new verification toolchain for reachability safety that combines the abstract interpreter GOBLINT with the witness validator WITCH. GOBLITCH integrates these two tools in a refinement loop where GOBLINT repeatedly builds an abstract reachability graph and WITCH decides whether an error path in the current graph corresponds to any real program execution. Experiments show that GOBLITCH produces many **False** results and no incorrect results. Moreover, in the category *C.ReachSafety*, it produces more **True** results than GOBLINT.

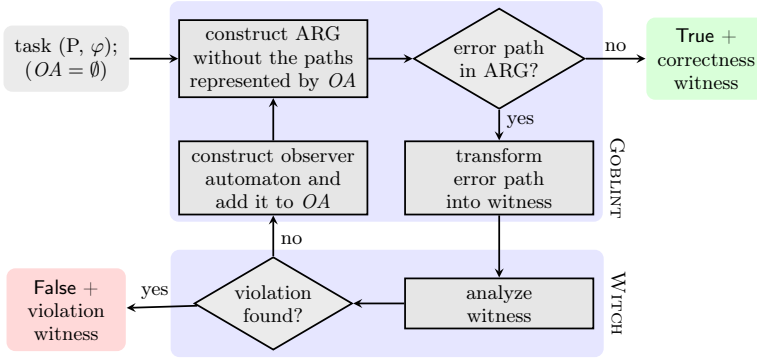
1 Verification Approach

GOBLITCH is a new verification tool for deciding whether an error location is reachable. Internally, it iteratively calls the abstract interpreter GOBLINT [15] and the violation witnesses validator WITCH [2] based on symbolic execution.

Being sound but incomplete, GOBLINT reports **True** when it can verify program correctness and **Unknown** otherwise, as it cannot determine whether a potential error is feasible. To support further analysis in the latter case, GOBLINT constructs an *abstract reachability graph* (ARG) [14]. The ARG nodes are abstract states, each representing a set of concrete states with the same program location, and the ARG edges over-approximate the program's transition relation. As a result, every concrete program execution is covered by a corresponding path in the ARG [8]. Paths in the ARG that lead to abstract states representing potential error locations are called *error paths*. GOBLITCH relies on WITCH to decide whether such error paths correspond to actual program executions.

GOBLITCH implements a component-based refinement loop [4]. It first runs GOBLINT, and if the program is proven safe, returns **True** with GOBLINT's correctness witness. Otherwise, GOBLINT represents the analysis results as an ARG, and searches it breadth-first for an error path. GOBLITCH converts that path into

* Jury member

Fig. 1: Refinement loop of $\hat{\text{GOBLITCH}}$.

a violation witness in format 2.0 [1]. The witness is constructed to allow arbitrary loop iterations to prevent inefficient loop unrolling during refinement. Therefore, the transformation ignores all edges whose source nodes lie within loops that the error path iterates at least once. Each remaining edge of the error path is turned into a witness segment containing only a *follow waypoint* of the following type: *branching* waypoints are used for edges corresponding to branching statements and *assumption* waypoints with constraint *true* for all other edges. The resulting witness thus represents all program executions that follow the acyclic parts of the error path and traverse the encountered loops in an arbitrary way.

To determine whether such a violation witness corresponds to a feasible program execution, $\hat{\text{GOBLITCH}}$ invokes WITCH . If the witness is confirmed, $\hat{\text{GOBLITCH}}$ returns **False** with the witness. Alternatively, during its exploration, WITCH may discover a violation target not matching the one in the witness. Then, $\hat{\text{GOBLITCH}}$ still returns **False**, but outputs a *new witness* generated by WITCH . For this, we extended WITCH with the ability to produce witnesses.

If WITCH decided that the witness does not represent any feasible execution, and it did not detect any other execution reaching an error location, $\hat{\text{GOBLITCH}}$ constructs an *observer automaton*. This automaton represents paths in ARG that do not correspond to any feasible execution. In general, the constructed observer automaton corresponds exactly to the original error path from the ARG on which the processed witness was based. However, if WITCH detects that a program location on this path is not reachable by any program execution leading to an error location, and the witness prefix leading to this location contained no branching waypoints, the automaton is modified to match the error paths containing the suffix starting at this unreachable location.

At this point, $\hat{\text{GOBLITCH}}$ starts the next iteration by invoking GOBLINT again. Using the observer automata accumulated so far, GOBLINT conducts a more precise analysis by excluding all paths represented by these automata. This refinement loop incrementally removes spurious error paths that do not correspond to any feasible program execution and converges toward real bugs when they exist. The refinement algorithm of $\hat{\text{GOBLITCH}}$ is also illustrated in Fig. 1.

Table 1: Solved tasks comparison between GOBLINT and $\hat{\text{GOBLITCH}}$.

(a) <i>C.ReachSafety</i>					(b) <i>C.unreach-call.SoftwareSystems-*</i>				
GOBLINT	$\hat{\text{GOBLITCH}}$				GOBLINT	$\hat{\text{GOBLITCH}}$			
	True	False	Unsolved	All		True	False	Unsolved	All
True	674	0	163	837	True	1,312	0	283	1,595
Unsolved	278	609	12,651	13,538	Unsolved	66	155	1,797	2,018
All	952	609	12,814	14,375	All	1,378	155	2,080	3,613

2 Software Architecture

GOBLINT is written in OCAML and uses its own fork of CIL [13] for parsing C preprocessed with GCC. It relies on APRON [9] for some relational analyses. The observer automata construction of Junker et al. [10] was adapted to GOBLINT as part of S. Saan’s M.Sc. thesis [14]. WITCH uses and extends parts of the SYMBIOTIC [3] framework, most notably the symbolic executor JETKLEE, which is a fork of KLEE [6] and utilizes the Z3 [12] solver. It relies on LLVM 14 [11].

3 Strengths and Weaknesses

$\hat{\text{GOBLITCH}}$ participated in the meta-categories *C.ReachSafety* and *C.SoftwareSystems*. Table 1 summarizes its results relative to GOBLINT, separately for *C.ReachSafety* (1a) and *C.unreach-call.SoftwareSystems-** (1b). In *C.SoftwareSystems*, only the *unreach-call* property is considered by $\hat{\text{GOBLITCH}}$, while other properties are handled exclusively by GOBLINT. $\hat{\text{GOBLITCH}}$ solved a total of 764 False tasks that GOBLINT cannot solve with its purely over-approximating analyses. In addition, refinement with WITCH enabled $\hat{\text{GOBLITCH}}$ to solve 344 further True tasks beyond what GOBLINT could solve even with its recent portfolio of increasingly precise analyses [16]. On the 1,986 tasks solved by both tools, $\hat{\text{GOBLITCH}}$ was slower in most cases; Figure 2 compares their CPU times. Nevertheless, $\hat{\text{GOBLITCH}}$ achieved higher scores in both categories: 6,552 points in *C.ReachSafety* (compared to 4,664 for GOBLINT) and 610 points in *C.SoftwareSystems* (compared to 456). Still, $\hat{\text{GOBLITCH}}$ does not yet exploit GOBLINT’s latest portfolio of analysis configurations, as integrating it with repeated WITCH-based refinements is challenging. Refinement is currently implemented without parallelization, enforced timeouts, advanced heuristics for path selection, or caching of WITCH results, which affects performance and limits scalability.

4 Setup and Configuration

The release archive contains the exact binary used in the competition running on Ubuntu 24.04. Sources of the two constituent components, GOBLINT and

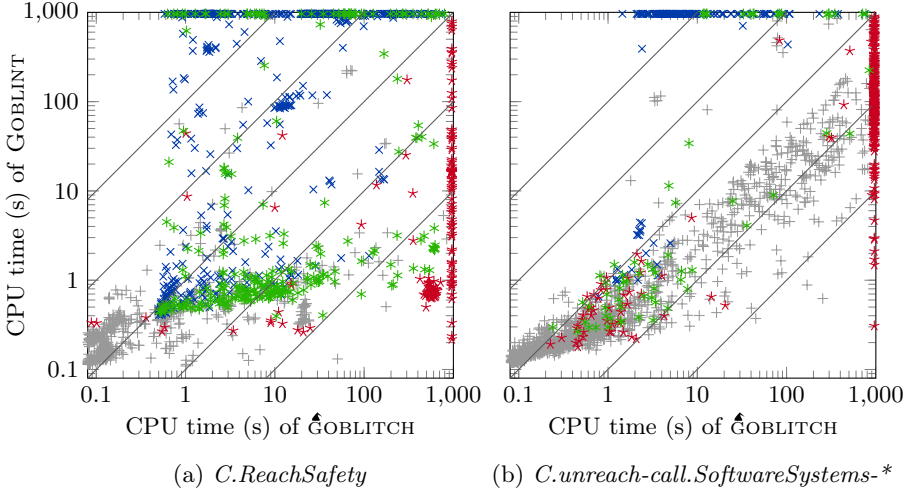


Fig. 2: CPU time comparison of $\hat{\text{GOBLITCH}}$ and GOBLINT. Gray (+) marks True tasks solved by both tools; green (*) those solved only by $\hat{\text{GOBLITCH}}$; red (x) those solved only by GOBLINT; blue (x) False tasks solved by $\hat{\text{GOBLITCH}}$.

WITCH, are available in separate public repositories.³ Building $\hat{\text{GOBLITCH}}$ from source⁴ is supported via the build script included in the GOBLINT repository.

$\hat{\text{GOBLITCH}}$'s SV-COMP invocation is defined by the `goblnt` tool-info module and corresponds to the following command:

```
./goblnt --conf conf/svcomp26.json \
  --set ana.specification property.prp \
  --set exp.architecture 64bit input.c
```

The configuration `conf/svcomp26.json` defines the set parameters. $\hat{\text{GOBLITCH}}$ follows the standard SV-COMP interface: the property is provided through `ana.specification`, and architecture via `exp.architecture` (32bit or 64bit).

5 Software Projects and Contributors

$\hat{\text{GOBLITCH}}$ is a part of the GOBLINT ecosystem and is developed and maintained by the Laboratory for Software Science at the University of Tartu and the Faculty of Informatics at Masaryk University. It participates in SV-COMP 2026 as an independent verification tool and integrates two MIT-licensed components: the GOBLINT analyzer (TU München and University of Tartu) and the WITCH witness validator (Masaryk University). All components of WITCH (Z3, JETKLEE, SYMBIOTIC) are distributed under permissive licenses. Development is open-source on GitHub with contributions from all involved institutions.

³ <https://github.com/goblnt/analyzer> and <https://github.com/ayazip/witch>

⁴ <https://github.com/goblnt/analyzer/releases/tag/goblitch-svcomp26>

Acknowledgments. P. Ayaziová and J. Strejček were supported by Horizon Europe CHES 101087529 and by the Czech Science Foundation project GA23-06506S. K. Holter, S. Saan, and V. Vojdani were supported by the European Union and the [Estonian Research Council](#) via projects PRG2764 and TEM-TA119.

Data Availability Statement. The complete SV-COMP 2026 [5] data set, including verification tasks, results, witnesses, and reproduction scripts, is available on the [competition website](#) [5]. The competition version of GOBLITCH is archived on Zenodo [7].

Bibliography

- [1] Ayaziová, P., Beyer, D., Rosenfeld, M.L., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Neele, T., Wijs, A. (eds.) Model Checking Software - 30th International Symposium, SPIN 2024, Luxembourg City, Luxembourg, April 8-9, 2024, Proceedings, Lecture Notes in Computer Science, vol. 14624, pp. 184–203, Springer (2024), https://doi.org/10.1007/978-3-031-66149-5_11
- [2] Ayaziová, P., Strejček, J.: Witch 3: Validation of violation witnesses in the witness format 2.0 (competition contribution). In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III, Lecture Notes in Computer Science, vol. 14572, pp. 341–346, Springer (2024), https://doi.org/10.1007/978-3-031-57256-2_18
- [3] Ayaziová, P., Jonáš, M., Mihalkovič, V., Sedláček, J., Strejček, J.: SYMBIOTIC 11: Predicate abstraction joins the party (competition contribution). In: Proc. TACAS (2), Lecture Notes in Computer Science, vol. 16506, Springer (2026)
- [4] Beyer, D., Haltermann, J., Lemberger, T., Wehrheim, H.: Decomposing software verification into off-the-shelf components: An application to CEGAR. In: 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pp. 536–548, ACM (2022), <https://doi.org/10.1145/3510003.3510064>
- [5] Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In: Proc. TACAS (2), Lecture Notes in Computer Science, vol. 16506, Springer (2026)
- [6] Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings, pp. 209–224, USENIX Association (2008), URL http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
- [7] Holter, K., Ayaziová, P., Saan, S.: Goblitch in SV-COMP 2026 (Nov 2025), <https://doi.org/10.5281/zenodo.17661717>

- [8] Holter, K., Hennoste, J.O., Lam, P., Saan, S., Vojdani, V.: Abstract debuggers: Exploring program behaviors using static analysis results. In: Edwards, J., Taeumel, M. (eds.) *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2024*, Pasadena, CA, USA, October 23–25, 2024, pp. 130–146, ACM (2024), <https://doi.org/10.1145/3689492.3690053>
- [9] Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) *Computer Aided Verification, 21st International Conference, CAV 2009*, Grenoble, France, June 26 - July 2, 2009. *Proceedings, Lecture Notes in Computer Science*, vol. 5643, pp. 661–667, Springer (2009), https://doi.org/10.1007/978-3-642-02658-4_52
- [10] Junker, M., Huuck, R., Fehnker, A., Knapp, A.: SMT-based false positive elimination in static program analysis. In: Aoki, T., Taguchi, K. (eds.) *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012*, Kyoto, Japan, November 12–16, 2012. *Proceedings, Lecture Notes in Computer Science*, vol. 7635, pp. 316–331, Springer (2012), https://doi.org/10.1007/978-3-642-34281-3_23
- [11] Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004)*, 20–24 March 2004, San Jose, CA, USA, pp. 75–88, IEEE Computer Society (2004), <https://doi.org/10.1109/CGO.2004.1281665>
- [12] de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. *Proceedings, Lecture Notes in Computer Science*, vol. 4963, pp. 337–340, Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_24
- [13] Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) *Compiler Construction, 11th International Conference, CC 2002*, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8–12, 2002. *Proceedings, Lecture Notes in Computer Science*, vol. 2304, pp. 213–228, Springer (2002), https://doi.org/10.1007/3-540-45937-5_16
- [14] Saan, S.: *Witness Generation for Data-flow Analysis*. Master’s thesis, University of Tartu (2020), URL <https://hdl.handle.net/10062/94066>
- [15] Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: Goblint: Abstract interpretation for memory safety and termination (competition contribution). In: Finkbeiner, B., Kovács, L. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024*, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Lux-

- embourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III, Lecture Notes in Computer Science, vol. 14572, pp. 381–386, Springer (2024), https://doi.org/10.1007/978-3-031-57256-2_25
- [16] Saan, S., Kocal, A.R., Petter, M., Holter, K., Erhard, J., Schwarz, M., Vojdani, V., Seidl, H.: GOBLINT: A portfolio for mixed flow-sensitive abstract interpretation (competition contribution). In: Proc. TACAS (2), Lecture Notes in Computer Science, vol. 16506, Springer (2026)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Hornix: From LLVM IR to Constrained Horn Clauses and Back (Competition Contribution)

Martin Blichá¹, Jan Kofroň¹, and Oliver Glitta

Charles University, Faculty of Mathematics and Physics, Czech Republic
martin.blichá@gmail.com, jan.kofron@d3s.mff.cuni.cz

Abstract. HORNIX is a software verification tool for C programs based on a translation first to LLVM intermediate representation (IR) and subsequently to constrained Horn clauses (CHCs). It extends LLVM with a transformation pass that constructs a system of CHCs from the LLVM IR representation and the safety property of the program. The resulting CHC system is then passed to a backend Horn solver, such as Eldarica, Golem, or Z3-Spacer. HORNIX is currently in an early stage of development and supports reasoning about safety properties of integer programs only.

Keywords: CHC · LLVM · Transformation pass

1 Verification Approach

HORNIX is a verification tool designed for programs written in C and C++. It is based on the LLVM Clang implementation [10]; in particular, it implements a custom transformation pass that produces a representation of the input program in the form of a system of constrained Horn clauses (CHCs) [4]. HORNIX is particularly useful for verifying basic assertions about variable states and program paths. The verification process consists of four main stages: (i) front-end compilation, (ii) intermediate representation (IR) preprocessing, (iii) CHC transformation, and (iv) CHC solving.

HORNIX does not prescribe any particular CHC solver as a backend; instead, any suitable CHC solver can be plugged in. For development and testing, we employed Z3 [11] and GOLEM [5], both of which are capable of solving CHC systems. The CHC output of the transformation pass is solver-independent, uses the SMT-LIB format [1], and does not rely on any solver-specific extensions.

HORNIX is inspired by other verification tools that harness the power of CHC solvers, namely SEAHORN [9], TRICERA [8], and KORN [7] for C programs. SEAHORN also leverages LLVM IR as the intermediate representation before translating the program to Horn clauses. While HORNIX is built on the same foundations as SEAHORN, the long-term goal of HORNIX is to develop novel encodings of programs and their properties into Horn clauses, as well as

M. Blichá—Jury member

to experiment with and evaluate new backend algorithms developed for CHC solving.

2 Software Architecture

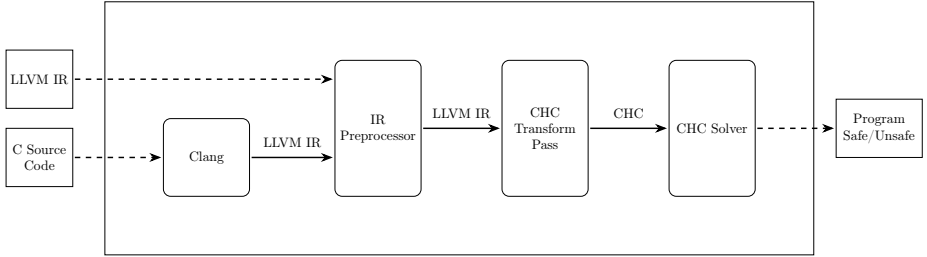


Fig. 1: HORNIX architecture

The architecture of HORNIX is depicted in Fig. 1. The tool is implemented in C++ and is accompanied by shell and Python scripts to facilitate easier usage and automation. The verification process is composed of four stages.

In the first stage, a frontend compiler (Clang) lowers the source code into LLVM Intermediate Representation (IR) [10]. None of the default Clang optimization pipelines are applied at this point.

In the second stage, the IR is transformed into a more verification-friendly form. Several carefully selected transformation passes are applied at the IR level with the aim of simplifying the code before CHC encoding. The most important pass is `mem2reg`, which promotes memory references to register references, effectively constructing an SSA version of the input program. This pass is followed by passes that simplify the control-flow graph (`simplifycfg`) and individual instructions (`instsimplify`). The preprocessing concludes with dead code elimination. We are also experimenting with custom IR transformations specifically aimed at reducing the complexity of the logical representation of the IR.

The third stage, CHC transformation, constitutes the core component of HORNIX. During this stage, the tool transforms the LLVM IR representation of the program into a corresponding CHC system. This transformation is implemented using the LLVM pass framework. The pass does not modify the IR, but generates Horn clauses from program instructions such that the safety of the program maps directly to the satisfiability of the CHC system. The transformation follows the approach described by Bjørner et al. [4]. In particular, for each basic block, two predicates are generated—one defining the variable values at the entry of the basic block, and the other defining the values at its exit. Next, the semantics of the basic block are captured by a clause transforming the entry values into the exit values. Further, for each control-flow edge between basic blocks, a clause capturing the connection is generated. This approach follows the

principle of *single block encoding* [3]. Moreover, to enable modular verification, for each function, a predicate is introduced representing the function summary (input-output relation). Next, a clause connecting the final basic block with the function summary is added. The summary predicate is then used to represent function calls at call sites.

In the final stage, an off-the-shelf CHC solver is employed to decide the satisfiability of the generated CHC system. HORNIX generates clauses in the CHC-COMP format [6], and any compliant solver can be used at this stage. Users can specify the CHC solver via command-line arguments.

Splitting the tool into well-defined stages provides significant flexibility for future improvements in both scope and performance. For example, instead of starting with a C program, HORNIX can also accept LLVM IR as input directly. In this case, the first stage is skipped and the IR is passed directly to the pre-processor. Thus, HORNIX can also be used to verify programs written in other languages that lower to LLVM IR during compilation. In the following stages, this modularity facilitates experimentation with new preprocessing passes and different CHC encodings. At the backend, HORNIX directly benefits from improvements in CHC solvers. Moreover, as different CHC solvers possess complementary strengths, we intend to explore multiple configurations for cooperative solving in the future. The modular architecture readily supports this.

3 Discussion of Strengths and Weaknesses of the Approach

Currently, only basic language features and data types are supported, along with a limited set of LLVM IR instructions (Fig. 2). The supported language features include variable assignments, conditional statements, numeric operations, loops, function calls, and logical operations over Boolean values. These features are handled using integer and Boolean data types only.

The verification process in HORNIX is centered on validating `assert` statements, which are required to be present in the input program; that is, HORNIX currently supports reasoning about reachability properties only. In C, assertions are provided by the `assert.h` library. Another requirement is the presence of the `main` function, as is customary in C applications.

Support for additional C language features, as well as C++ programs—which would imply support for a broader set of LLVM IR instructions, most notably pointers and related concepts such as global variables and string manipulation—is left for future work. Furthermore, while CHC solvers are capable of producing witnesses at the CHC level [12,13], the transformation of a computed witness back to the LLVM and C levels has not yet been implemented and is also planned as future work. For this reason, HORNIX currently produces empty witnesses only.

In the 2026 edition of SV-COMP, HORNIX participated in three subcategories of the `C.unreach-call` category: `C.unreach-call.Loops`, `C.unreach-call.Recursive`, and `C.unreach-call.XCSP`. It used Golem [5] as the backend CHC

Add, And, Br, Call, CallBr, ICmp, IndirectBr, Load, Mul, Or, PHI, Ret, SDiv, Select, SExt, SRem, Store, Sub, Switch, Trunc, UDiv, Unreachable, URem, Xor, ZExt

Fig. 2: List of supported LLVM IR instructions

solver. GOLEM was configured to run two of its engines in parallel: *pdkind* and *spacer*. In the **Recursive** subcategory, HORNIX placed 7th out of 34 tools. Interestingly, under a hypothetical timeout of 10 seconds, HORNIX would become the best-performing tool in this subcategory. We believe that with support for pointers, HORNIX has the potential to win this subcategory in the future.

In the **XCSP** subcategory, HORNIX placed 8th out of 33 tools, demonstrating significant potential. Finally, it placed 18th out of 35 tools in the **Loops** subcategory, where many benchmarks contain features not yet supported by HORNIX. Notably, in both the **Recursive** and **XCSP** subcategories, HORNIX outperformed the other CHC-based tools KORN and THORN. The complete results are available in the SV-COMP 2026 report [2].

4 Tool Setup and Configuration

The project is available as a GitHub repository¹. The **README** file provides instructions for compiling and running the tool, including information on installing dependencies and their tested versions, in particular compatible versions of LLVM.

The HORNIX tool can be executed with a single parameter specifying the input C or LLVM IR file to be verified. In addition, it offers several command-line options that allow users to customize its behavior.

5 Software Project and Contributors

The principal HORNIX developers are Oliver Glitta and Martin Blich, while Jan Kofron has so far provided minor implementation contributions. The tool is distributed under the Apache License 2.0².

Acknowledgements. The work was partially supported by the Czech Science Foundation project no. 23-06506S, and by the Ministry of Education, Youth and Sports of the Czech Republic, the ERC CZ project no. LL2325.

Data-Availability Statement. The version of HORNIX corresponding to this paper is available in the Zenodo repository under the DOI: <https://doi.org/10.5281/zenodo.17638644>.

¹ <https://github.com/d3sformal/hornix>

² <https://www.apache.org/licenses/LICENSE-2.0.html>

References

1. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Tech. rep., Department of Computer Science, The University of Iowa (2010), available at www.SMT-LIB.org
2. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
3. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., University, S.F., Sebastiani, R.: Software model checking via large-block encoding. In: 2009 Formal Methods in Computer-Aided Design. pp. 25–32 (2009). <https://doi.org/10.1109/FMCAD.2009.5351147>
4. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn Clause Solvers for Program Verification, pp. 24–51. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
5. Blicha, M., Britikov, K., Sharygina, N.: Golem: a flexible and efficient solver for constrained Horn clauses. *Form Methods Syst Des* **67**, 143–160 (2025), <https://doi.org/10.1007/s10703-025-00470-9>
6. CHC-COMP format. <https://chc-comp.github.io/format.html>
7. Ernst, G.: Korn—Software Verification with Horn Clauses (Competition Contribution). In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 559–564. Springer-Verlag, Berlin, Heidelberg (2023)
8. Esen, Z., Rümmer, P.: TRICERA: Verifying C programs using the theory of heaps. In: Griggio, A., Rungta, N. (eds.) Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design – FMCAD 2022. pp. 360–391. TU Wien Academic Press (2022)
9. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 343–361. Springer International Publishing, Cham (2015)
10. The LLVM Compiler Infrastructure. <https://llvm.org>
11. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
12. Otoni, R., Blicha, M., Barandiaran Rivera, M., Eugster, P., Kofroň, J., Sharygina, N.: Unsatisfiability proofs for Horn solving. In: Gurfinkel, A., Heule, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 67–87. Springer Nature Switzerland, Cham (2025)
13. Otoni, R., Blicha, M., Eugster, P., Sharygina, N.: Validation of CHC satisfiability with ATHENA. *Formal Aspects of Computing* **37**(4) (Oct 2025)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





IEKKĚ: A SAT-Based Bounded-Round Verifier for Multi-Threaded Programs (Competition Contribution)

Paolo Di Biase^{1,5} * **, Bernd Fischer² , Salvatore La Torre³ , Peter Schrammel^{4,6} , and Gennaro Parlato⁵

¹ Gran Sasso Science Institute (GSSI), L'Aquila, Italy, paolo.dibiase@gssi.it

² Stellenbosch University, Stellenbosch, South Africa, bfischer@cs.sun.ac.za

³ University of Salerno, Fisciano (SA), Italy, slatorre@unisa.it

⁴ University of Sussex, Brighton, UK, p.schrammel@sussex.ac.uk

⁵ University of Molise, Pesche, Italy, gennaro.parlato@unimol.it

⁶ Diffblue Ltd, UK

Abstract. IEKKĚ is a sequentialization-inspired bounded model checking (BMC) tool for verifying multi-threaded C programs. It is derived from DEAGLE and reuses the CBMC front-end for parsing, goto-program construction, loop unwinding, and SSA-based formula generation. IEKKĚ differs from both CBMC and DEAGLE in its treatment of shared-memory interactions: it replaces the classic partial-order memory constraints with a *bounded-round* encoding inspired by lazy sequentialization. The resulting shared-memory constraints are *linear* in the number of memory events of the bounded program and linear in the round bound k , avoiding the quadratic/cubic growth typical of partial-order encodings. Because the concurrency constraints are compact and purely propositional, IEKKĚ relies on an off-the-shelf SAT solver (e.g., MINISAT) rather than a dedicated ordering-theory solver. IEKKĚ supports reachability, assertion and data-race checking, and produces SV-COMP witnesses for violations.

Keywords: Program verification · Bounded model checking · Concurrency · SAT.

1 Verification Approach

Background. For shared-memory concurrency, BMC tools must connect reads and writes across threads in a way that corresponds to a consistent execution. Classical encodings, including the partial-order approach used in CBMC, introduce constraints over event orders (e.g., happens-before, coherence) and enforce consistency by global properties such as acyclicity [1,12]. On memory-intensive instances, these constraints can dominate the formula size.

* Jury member/Corresponding author.

** Work performed while at the University of Molise.

Bounded-round semantics. IEKKĚ (pronounced 'jekkə) under-approximates the interleaving space by restricting the analysis to executions that can be partitioned into at most k rounds. Intuitively, in each round threads execute in a fixed global order; each thread executes a (possibly empty) contiguous segment of its SSA steps and then yields to the next thread. This scheduling discipline is inspired by lazy sequentialization (e.g., LAZY-CSEQ) [9,11] and is effective for bug finding because many concurrency bugs manifest within a small number of context-switches [13]. LAZY-CSEQ's sequentialization is also used in [8,10,5,14,15].

Bounded-round memory constraints. For a fixed k , IEKKĚ introduces variables that represent (i) the per-thread *context-switch points* delimiting the segments executed by the thread in each round, and (ii) the *round assignment* of memory events. Shared-memory writes update the per-round snapshots of the shared variables; shared-memory reads select the value of the latest visible write according to the round and thread order. Crucially, the number of additional constraints is *linear* in (a) the number of shared-memory events in the bounded program and (b) the round bound k .

Relation to Lazy-CSeq and CBMC/DEAGLE. IEKKĚ is designed to inherit the strengths of both LAZY-CSEQ and CBMC while addressing key weaknesses of each. From LAZY-CSEQ, IEKKĚ inherits the idea of bounding concurrency by a small number of rounds and the resulting compact treatment of shared-memory interactions. However, unlike source-level lazy sequentialization, IEKKĚ does *not* replicate each thread's code for every round. Instead, it builds on CBMC's SSA-based front-end and introduces round/context-switch-point variables directly at the formula level, keeping a *single* symbolic copy of the program while still restricting schedules to at most k rounds. From CBMC, IEKKĚ inherits the mature parsing, unwinding, and SSA generation pipeline, but it replaces CBMC's resp. DEAGLE's partial-order-style shared-memory constraints—which can grow quadratically or worse in the number of events—with its own bounded-round memory constraints that are linear in the number of shared-memory events and in the round bound k . CBMC and DEAGLE both support weak memory models while IEKKĚ currently supports only sequential consistency.

Why SAT suffices. DEAGLE tries to address the scalability issues of large partial-order encodings by integrating a specialized solver component to reason about ordering consistency [6,7]. In contrast, IEKKĚ reduces the blow-up at the source: bounded-round constraints make shared-memory consistency largely explicit in the schedule (i.e., round and thread order), yielding a compact propositional encoding that can be handled effectively by standard CDCL SAT solvers such as MINISAT [4]. Moreover, the round bound k provides a smooth trade-off between compactness and coverage. In the extreme, choosing k large enough (e.g., $k \geq |E|$, where E is the set of shared-memory events in the bounded program) allows IEKKĚ to simulate executions with as many context-switch opportunities as there are memory events, and thus to capture the same read/write interaction patterns as the existing CBMC/DEAGLE encodings on the bounded program. In this worst

case, the size of the shared-memory constraints becomes $O(|E| \cdot k) = O(|E|^2)$, still avoiding the cubic growth that can arise in certain partial-order formulations.

Supported properties. IEKKĚ supports all reachability checks supported by CBMC (e.g., assertion violations, `unreach-call`) and concurrency-specific checks. For data races, IEKKĚ searches for executions witnessing conflicting unsynchronized accesses and reports them as violations.

2 Software Architecture

Implementation lineage. IEKKĚ inherits from but is not built directly on top of CBMC. Instead, it is derived from DEAGLE, which already incorporates several concurrency-oriented optimizations in the encoding of thread functions (partly inherited from Yogar-CBMC). IEKKĚ reuses this optimized CBMC/DEAGLE infrastructure for parsing, unwinding, and SSA generation, but replaces DEAGLE’s partial-order/ordering-theory layer with our bounded-round shared-memory constraints. IEKKĚ’s pipeline thus comprises the following steps:

- **Front-end.** This step includes preprocessing and parsing of C; construction of a goto-program; modeling of common concurrency primitives (e.g., Pthreads); loop unwinding up to the user-provided bound; and the SSA generation for thread-local control/data flow [12].
- **Concurrency encoding.** This step replaces DEAGLE’s partial-order layer with new bounded-round shared-memory constraints. The encoding is generated natively at the formula level (rather than via source-to-source sequentialization), which enables a tight integration with the existing SSA pipeline.
- **Solving.** The resulting propositional formula is discharged by an off-the-shelf SAT solver; we use MINISAT as default solver.
- **Counterexamples and witnesses.** IEKKĚ includes in the constructed SSA form the mapping between the SSA variables and the original program variables, following the style of CBMC/DEAGLE. Thus, for violated properties, IEKKĚ can reuse the counterexample generation and trace reporting infrastructure of the CBMC/DEAGLE tool chain.

IEKKĚ is implemented in C++, given its CBMC-derived architecture and codebase.

3 Strengths and Weaknesses

Strengths. On unsafe benchmarks where violations occur within a few rounds, IEKKĚ is competitive while often using substantially less time and memory than partial-order approaches such as DEAGLE and CBMC. On benchmarks with many memory events, the linear growth in events and rounds avoids the cubic blow-up typical of classic partial-order encodings, improving practical scalability.

Weaknesses. IEKKĚ uses a BMC approach; in addition to the bounds on loop unwinding and recursion depth, it introduces a bound on the number of rounds. In contrast to the former two, IEKKĚ currently does not automatically increase the round bound, which must be specified by the user; if a bug requires more rounds for discovery than specified, the tool will not report it. The approach is thus an under-approximation and is more specialized to find bugs.

Results. IEKKĚ scored 3428 points, earning the bronze medal in the *Concurrency* category [2]. It identified 308 bugs with confirmed witnesses and 216 without. IEKKĚ gave incorrect answers for 44 safe benchmarks due to minor implementation bugs and misclassified 22 unsafe benchmarks as safe: 3 of these were caused by DEAGLE issues, while the rest were due to considering only 3 rounds. DEAGLE correctly solved 40 of these 66 benchmarks. IEKKĚ performed well overall, except in the **no-data-race** sub-category, solving only 646 out of 1030 benchmarks.

4 Tool Setup and Configuration

Installation. IEKKĚ installation is identical to that of DEAGLE; detailed instructions are provided in the repository **README**.

Parameters. IEKKĚ reuses CBMC/DEAGLE’s standard parameters (i.e., architecture bit-width, unwinding, property selection). The maximum number of computation rounds is set via `--lazy-c-seq-rounds`; in the competition, we simply use a fixed value $k = 3$, since real bugs often manifest within 3 rounds; however, we can improve this by an iterative deepening scheme. Property selection is via dedicated parameters. For example, the **no-overflow** property requires the additional parameters `--no-assertions`, `--signed-overflow-check`, and `--unsigned-overflow-check`. We refer to the Zenodo archive for further details.

Participation. IEKKĚ participates only in the *Concurrency* category. We support the following properties: **no-data-race**, **no-overflow**, **unreach-call**, and **valid-memsafety**.

Usage. A typical invocation is:

```
deagle_exe file.c --lazy-c-seq-rounds 3 --unwind 9 <other opts>
```

5 Software Project

IEKKĚ is developed by the authors based on DEAGLE 4.1.0 and its corresponding CProver/CBMC infrastructure, using MINISAT as default SAT backend. IEKKĚ is licensed under GPLv3; see the repository for the exact competition package, third-party licenses, and build instructions.

Acknowledgements

We thank the developers and maintainers of CBMC, MINISAT, and DEAGLE for making their tools available and for their contributions to the verification community. This work was supported by the Erasmus+ International Credit Mobility Project 2024-1-IT02-KA171-HED-000224869, by INDAM-GNCS 2024-2025, FARB 2022–25 grants of Università degli Studi di Salerno, and by AWS Amazon Research Awards. Paolo Di Biase was supported by a Research Scholarship (Borsa di Ricerca) from the University of Molise.

Data Availability Statement

The tool is open-source and developed in the LAZY-PO repository:

<https://github.com/paolo-di-biase/Lazy-PO>

The SV-COMP submission is based on commit 50ddac2. The competition snapshot is also archived on Zenodo [3].

References

1. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science*, vol. 8044, pp. 141–157. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_9
2. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In: *Proc. TACAS (2). LNCS 16506*, Springer (2026)
3. Di Biase, P., Fischer, B., La Torre, S., Schrammel, P., Parlato, G.: Iekkë artifact. Zenodo (2025). <https://doi.org/10.5281/zenodo.17708300>
4. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. Lecture Notes in Computer Science*, vol. 2919, pp. 502–518. Springer (2003). https://doi.org/10.1007/978-3-540-24605-3_37
5. Fischer, B., Garbi, G., La Torre, S., Parlato, G., Schrammel, P.: Static data race detection via lazy sequentialization. In: Castañeda, A., Enea, C., Gupta, N. (eds.) *Networked Systems - 12th International Conference, NETYS 2024, Rabat, Morocco, May 29-31, 2024, Proceedings. Lecture Notes in Computer Science*, vol. 14783, pp. 124–141. Springer (2024). https://doi.org/10.1007/978-3-031-67321-4_8
6. He, F., Sun, Z., Fan, H.: Satisfiability modulo ordering consistency theory for multi-threaded program verification. In: Freund, S.N., Yahav, E. (eds.) *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. pp. 1264–1279. ACM (2021). <https://doi.org/10.1145/3453483.3454108>

7. He, F., Sun, Z., Fan, H.: Deagle: An smt-based verifier for multi-threaded programs (competition contribution). In: Fisman, D., Rosu, G. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*. *Lecture Notes in Computer Science*, vol. 13244, pp. 424–428. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_25
8. Inverso, O., Nguyen, T.L., Fischer, B., La Torre, S., Parlato, G.: Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In: Cohen, M.B., Grunske, L., Whalen, M. (eds.) *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. pp. 807–812. IEEE Computer Society (2015). <https://doi.org/10.1109/ASE.2015.108>
9. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded model checking of multi-threaded C programs via lazy sequentialization. In: Biere, A., Bloem, R. (eds.) *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science*, vol. 8559, pp. 585–602. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_39
10. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Lazy-cseq: A lazy sequentialization tool for C - (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Lecture Notes in Computer Science*, vol. 8413, pp. 398–401. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_29
11. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded verification of multi-threaded programs via lazy sequentialization. *ACM Trans. Program. Lang. Syst.* **44**(1), 1:1–1:50 (2022). <https://doi.org/10.1145/3478536>
12. Kroening, D., Schrammel, P., Tautschnig, M.: CBMC: the C bounded model checker. *CoRR abs/2302.02384* (2023). <https://doi.org/10.48550/ARXIV.2302.02384>
13. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Ferrante, J., McKinley, K.S. (eds.) *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. pp. 446–455. ACM (2007). <https://doi.org/10.1145/1250734.1250785>
14. Nguyen, T.L., Schrammel, P., Fischer, B., La Torre, S., Parlato, G.: Parallel bug-finding in concurrent programs via reduced interleaving instances. In: Rosu, G., Penta, M.D., Nguyen, T.N. (eds.) *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. pp. 753–764. IEEE Computer Society (2017). <https://doi.org/10.1109/ASE.2017.8115686>
15. Tomasco, E., Nguyen, T.L., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Lazy sequentialization for TSO and PSO via shared memory abstractions. In: Piskac, R., Talupur, M. (eds.) *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*. pp. 193–200. IEEE (2016). <https://doi.org/10.1109/FMCAD.2016.7886679>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





JLiSA: The Java Frontend of the Library for Static Analysis (Competition Contribution)

Vincenzo Arceri^{1*}, Luca Negrini^{2*}, Giacomo Zanatta^{2*†}, Filippo Bianchi^{1‡}, Teodors Lisovenko^{2‡}, Luca Olivieri^{2‡}, and Pietro Ferrara²

¹ University of Parma, Italy

`vincenzo.arceri@unipr.it`, `filippo.bianchi5@studenti.unipr.it`

² Ca' Foscari University of Venice, Italy

`{luca.negrini, giacomo.zanatta, teodors.lisovenko, luca.olivieri, pietro.ferrara}@unive.it`

Abstract. JLiSA is the extension to the Java programming language of LiSA, an analysis engine that works on a generic and extensible control flow graph representation of the program to analyze. LiSA implements several standard abstract domains and analyses aimed at approximating numerical values, strings, and heap structures. At the end of the analysis, it produces an abstract state for each program point. Then, checkers produce warnings indicating whether a property of interest is respected. JLiSA provides a front-end to translate Java programs into the internal LiSA control flow graph representation, the semantics of various parts of the Java standard library, and checkers to verify assertions and detect whether exceptions might be thrown and not caught. This paper presents our first participation in SV-COMP in the Java category, where we achieved 3rd place.

1 Verification Approach

JLiSA relies on LiSA [9,13,11] (Library for Static Analysis), a Java library that provides a complete infrastructure for the development of static analyzers based on abstract interpretation [6,7]. In particular, LiSA implements several standard components of abstract interpretation-based analyzers, including an extensible control flow graph (CFG) representation, a standard analysis framework for developing new static analyses, and fixpoint algorithms on LiSA CFGs. LiSA has been extended and applied to a large variety of properties, programming languages and contexts: Go smart contracts and blockchain software [21,22,17,19,18,16,14], Python robotic software [25,24], microservices [26], Michelson [20,15] and EVM bytecode [3,2], as well as for teaching purposes [12]. JLiSA builds on top of LiSA to analyze Java programs. In particular, it consists

* Equal contribution

† Jury member

‡ Equal contribution

of a front-end that translates Java programs into LiSA’s internal control flow graph representation. This representation is enriched with the semantics of a subset of the Java standard library, corresponding to the classes and methods exercised by the AutoStub category of the SV-COMP benchmarks³. Then, the parsed code is passed to LiSA, along with Java-specific semantic algorithms (e.g., class hierarchy traversal, call target matching, ...) and the analysis configuration. The latter includes, among others, the list of abstract domains to execute and the property checkers that will issue warnings. These can be either existing components provided by LiSA or custom ones defined in JLiSA. Regarding SV-COMP 2026 [4], JLiSA was configured with the following settings:

- *Analysis Settings.* We configured the interprocedural analysis, which computes a program-wide fixpoint, to start in the `main` methods and use a call-string-based approach for managing contexts at call sites [23], with a maximum depth of 150 nested calls. After evaluation, we found that this depth is well-suited for SV-COMP benchmarks, providing a good trade-off between precision and performance. For call chains longer than 150, the analysis resorts to an over-approximation (the top abstract state). Since call-string analysis alone does not resolve Java’s virtual method targets, 0-CFA [10] was used to construct the call graph.
- *State Settings.* The state of the analysis was structured according to Ferrara et al. [8], employing: (i) a field-sensitive program-point based heap abstraction [1] to track objects and their fields, (ii) a reduced product between a constant propagation analysis (tracking numerical, string, and Boolean values) and the interval domain [6,7], for abstracting values, and (iii) type inference for considering the runtime types of expressions. The state was further wrapped into a *reachability* analysis, that distinguishes definitely reachable instructions from possibly reachable or unreachable ones relying on the traversed conditions; for instance, the true branch of an if-then-else statement is definitely reachable only if (i) the condition is reachable, and (ii) the condition is always satisfied — if the latter is instead sometimes satisfied, the branch is considered possibly reachable.

After the analysis, two property checkers were executed, one for runtime exceptions and one for failing asserts.

1.1 Tool pipeline and Software Architecture

The BenchExec framework [5] orchestrates the execution of our tool within the SV-COMP benchmarking infrastructure. A dedicated CLI tool preprocesses the Java benchmarks by translating property specifications and configuration files into JLiSA inputs. This preprocessing step sets the *checks options*, such as runtime safety and assertion violation checks, and other *configuration options*, according to the properties to be verified and the Java source files under analysis.

³ <https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp26/java/autostub>

At the end of execution, JLiSA frontend may return an analysis result, containing warnings. The CLI tool then collects and post-processes the analysis results. Specifically, it uses the property file to determine which warnings to extract and process from JLiSA’s analysis output and the selected warnings are used to produce a final report in the format required by SV-COMP. In particular, there are two main generic families of warnings (possible or definite). In addition, warnings about assertions report whether an assertion holds or does not. JLiSA reports that a test case is verified if either no warning is produced, or only definite warnings about assertions that hold are produced. Instead, it reports that a test case is not verified if only definite warnings about thrown exceptions or assertions that do not hold are produced. In all other cases (e.g., assertions that may or may not hold), JLiSA returns unknown to avoid false positives and false negatives. LiSA and JLiSA are both developed in Java (version 11 and 17, respectively), and rely on several standard libraries (such as log4j, JUnit, and reflections). JLiSA parses Java code through the Eclipse Java Development Tools. Instead, the CLI tool is written in Python and relies on PyYAML to read YAML files, and Typer to build the CLI application.

1.2 JLiSA Architecture

Figure 1 draws the overall architecture and an in-depth overview of JLiSA frontend components.

As shown in Figure 1, JLiSA (i) transforms the Java source code into its internal representation (middle left), (ii) performs a fixpoint computation of the semantics over the domains specified in the configuration (center), and (iii) produces an abstract state for each program point. All the standard components of the analysis rely on the LiSA library (grey background), while other components have been extended and customized in JLiSA (white background).

$\top$ and $\perp$ are special elements, representing *any value* and *no possible value*, respectively. Instead, CP, INT, R, DR denote *constant propagation* and *interval* abstract domains, *reachability*, and *definitely reachable*, respectively.

After the analysis, two semantic checks are executed to produce the final report (bottom right). The first verifies the validity of assert statements by reasoning about (i) their reachability and (ii) the satisfiability of their conditions. The second identifies statements that may throw runtime exceptions and determines whether such exceptions are properly caught. The two checks are always executed simultaneously, and JLiSA is unaware of which property is under investigation; therefore, both kinds of warnings may appear in its output. It is the CLI tool that uses the property file to determine which warnings to extract and process from JLiSA’s analysis output.

2 Strengths and Weaknesses

The main strength and weakness of JLiSA are in its foundation in abstract interpretation theory. We strongly believe in the need for (coarse) approximation

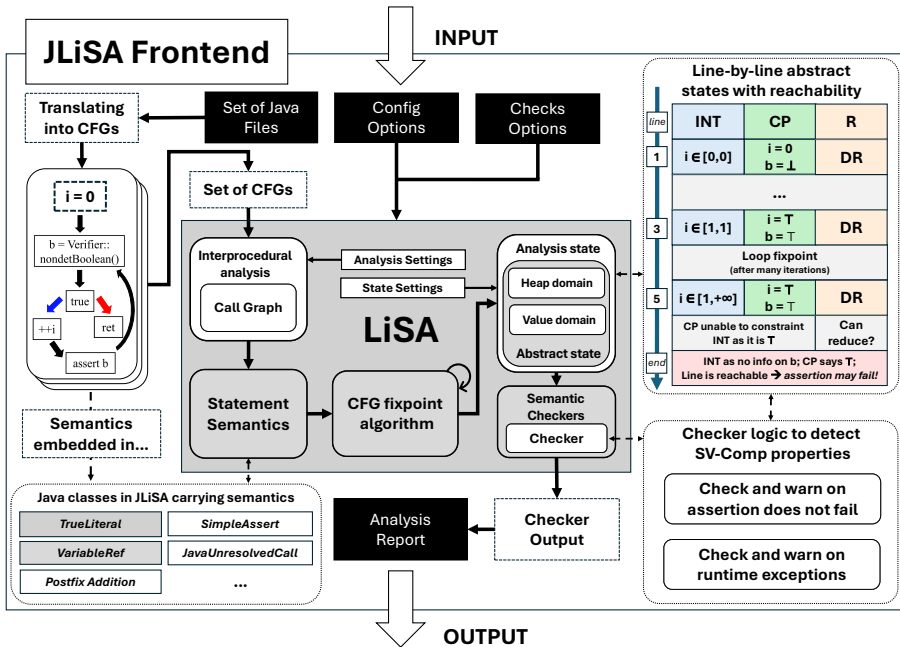


Fig. 1: JLiSA architecture

in order to make sound static analysis feasible. In practice, this means that (i) we can develop a wide variety of abstractions, modifying them based on our needs (strength), and (ii) we might produce false alarms (weakness). Point (i) gave us a strong advantage over other tools, since we could quickly adapt and modify the analyses we apply to the SV-COMP benchmark, especially when test cases were added (which was quite the case this year in the Java category). Point (ii) penalizes us since false alarms in SV-COMP are highly penalized. Generally speaking, a precision of 90% (that is, a pretty good precision for any generic abstract interpretation-based static analyzer) would be an excellent result, but this would end up in a negative score at SV-COMP (as a single false alarm incurs a -16 penalty versus a $+2$ benefit for a true alarm). To mitigate this weakness, we carefully post-processed the alarms generated by our analysis, as described in Section 1.2, and reported an alarms only when we were fully confident that it was a true positive.

The scores obtained by JLiSA at SVCOMP properly reflect our design choices. In particular, more than 95% of the points were awarded for correct true reports. Since the rules we applied to raise warnings were quite strict to avoid incorrect false reports, JLiSA did not produce any, thanks also to our conservative approach detailed in Section 1 for handling unproven cases. In addition, since JLiSA is sound, no incorrect true report was produced.

Data Availability Statement. The Unix binary of JLiSA that participated in SV-COMP 2026 is available at <https://doi.org/10.5281/zenodo.17609338>. The source codes are publicly available at <https://github.com/lisa-analyzer/jlisa> and <https://github.com/lisa-analyzer/lisa> for JLiSA and LiSA, respectively. JLiSA Gradle’s configuration defines its dependency on specific LiSA versions. Both projects are released with an MIT license. The competition artifact was generated from commit ID ce67adb (tag svcomp-2026 of the JLiSA repository).⁴ The JLiSA executable can be obtained either by downloading the competition archive or by building it from source as described in the README⁵.

Acknowledgments. Work partially supported by SERICS (PE00000014 - CUP H73C2200089001), iNEST (ECS00000043 – CUP H43C22000540006) projects funded by PNRR NextGeneration EU; and by the EU—NGEU National Sustainable Mobility Center (CN000000023), Italian Ministry of University and Research Decree n. 1033—17/06/2022 (Spoke 10), Regione Veneto: RIR project SATCO (CUP D19J24000750007), RIR project SUPREME (CUP D19J24000810007), and by Bando di Ateneo 2024 per la Ricerca, funded by University of Parma (FIL_2024_PROGETTILB.IOTTI - CUP D93C24001250005).

This work has benefited from the participation of the first and last authors, as well as the SV-COMP competition chair Dirk Beyer, in the Dagstuhl Seminar 25421 ”Sound Static Program Analysis in Modern Software Engineering”.

References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Phd thesis, University of Copenhagen, DIKU (1994)
2. Arceri, V., Merenda, S.M., Dolcetti, G., Negrini, L., Olivieri, L., Zaffanella, E.: Towards a sound construction of evm bytecode control-flow graphs. In: Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs. p. 11–16. FTfJP 2024, Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3678721.3686227>
3. Arceri, V., Merenda, S.M., Negrini, L., Olivieri, L., Zaffanella, E.: Evmlisa: Sound static control-flow graph construction for evm bytecode. Blockchain: Research and Applications p. 100384 (2025). <https://doi.org/https://doi.org/10.1016/j.bcra.2025.100384>
4. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
5. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. International Journal on Software Tools for Technology Transfer **21**(1), 1–29 (Feb 2019). <https://doi.org/10.1007/s10009-017-0469-y>
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings

⁴ In the Zenodo artifact, we explicitly added the commit ID to the JLiSA version (v0.2-ce67adbda) to distinguish it from other versions. Locally built versions may appear without the commit ID.

⁵ <https://github.com/lisa-analyzer/sv-comp#build-sv-comp-2026-executable>

- of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 238–252. POPL '77, Association for Computing Machinery, New York, NY, USA (1977). <https://doi.org/10.1145/512950.512973>
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979. pp. 269–282. ACM Press (1979). <https://doi.org/10.1145/567752.567778>
 8. Ferrara, P.: A generic framework for heap and value analyses of object-oriented programming languages. *Theoretical Computer Science* **631**, 43–72 (Jun 2016). <https://doi.org/10.1016/j.tcs.2016.04.001>
 9. Ferrara, P., Negrini, L., Arceri, V., Cortesi, A.: Static analysis for dummies: experiencing lisa. In: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. p. 1–6. SOAP 2021, Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3460946.3464316>
 10. Grove, D., DeFouw, G., Dean, J., Chambers, C.: Call graph construction in object-oriented languages. In: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. pp. 108–124. OOPSLA '97, Association for Computing Machinery, New York, NY, USA (Oct 1997). <https://doi.org/10.1145/263698.264352>
 11. Negrini, L.: A generic framework for multilanguage analysis. Phd thesis, Università Ca' Foscari Venezia (2023)
 12. Negrini, L., Arceri, V., Olivieri, L., Cortesi, A., Ferrara, P.: Teaching through practice: Advanced static analysis with lisa. In: Sekerinski, E., Ribeiro, L. (eds.) Formal Methods Teaching - 6th Formal Methods Teaching Workshop, FMTea 2024, Milan, Italy, September 10, 2024, Proceedings. Lecture Notes in Computer Science, vol. 14939, pp. 43–57. Springer (2024). https://doi.org/10.1007/978-3-031-71379-8_3
 13. Negrini, L., Ferrara, P., Arceri, V., Cortesi, A.: Lisa: A generic framework for multilanguage static analysis. In: Arceri, V., Cortesi, A., Ferrara, P., Oliaro, M. (eds.) Challenges of Software Verification, Intelligent Systems Reference Library, vol. 238, pp. 19–42. Springer (2023). https://doi.org/10.1007/978-981-19-9601-6_2
 14. Olivieri, L.: Detection of cross-channel invocation risks in hyperledger fabric. In: 2025 IEEE 36th International Symposium on Software Reliability Engineering (IS-SRE). pp. 107–118 (2025). <https://doi.org/10.1109/ISSRE66568.2025.00023>
 15. Olivieri, L., Jensen, T., Negrini, L., Spoto, F.: Michelsonlisa: A static analyzer for tezos. In: 2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops). pp. 80–85 (2023). <https://doi.org/10.1109/PerComWorkshops56833.2023.10150247>
 16. Olivieri, L., Negrini, L.: Don't panic: Error handling patterns in go smart contracts and blockchain software. In: 2025 7th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS). pp. 1–9 (2025). <https://doi.org/10.1109/BRAINS67003.2025.11302935>
 17. Olivieri, L., Negrini, L., Arceri, V., Chachar, B., Ferrara, P., Cortesi, A.: Detection of phantom reads in hyperledger fabric. *IEEE Access* **12**, 80687–80697 (2024). <https://doi.org/10.1109/ACCESS.2024.3410019>
 18. Olivieri, L., Negrini, L., Arceri, V., Ferrara, P., Cortesi, A.: Detection of read-write issues in hyperledger fabric smart contracts. In: Hong, J., Battiato, S., Esposito, C., Park, J.W., Przybyłek, A. (eds.) Proceedings of the 40th ACM/SIGAPP

- Symposium on Applied Computing, SAC 2025, Catania International Airport, Catania, Italy, 31 March 2025 - 4 April 2025. pp. 329–337. ACM (2025). <https://doi.org/10.1145/3672608.3707721>
19. Olivieri, L., Negrini, L., Arceri, V., Ferrara, P., Cortesi, A., Spoto, F.: Static detection of untrusted cross-contract invocations in go smart contracts. In: Hong, J., Battiato, S., Esposito, C., Park, J.W., Przybylek, A. (eds.) Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing, SAC 2025, Catania International Airport, Catania, Italy, 31 March 2025 - 4 April 2025. pp. 338–347. ACM (2025). <https://doi.org/10.1145/3672608.3707728>
 20. Olivieri, L., Negrini, L., Arceri, V., Jensen, T.P., Spoto, F.: Design and implementation of static analyses for tezos smart contracts. *Distributed Ledger Technol. Res. Pract.* **4**(2), 13:1–13:23 (2025). <https://doi.org/10.1145/3643567>
 21. Olivieri, L., Negrini, L., Arceri, V., Tagliaferro, F., Ferrara, P., Cortesi, A., Spoto, F.: Information flow analysis for detecting non-determinism in blockchain. In: Ali, K., Salvaneschi, G. (eds.) 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17–21, 2023, Seattle, Washington, United States. *LIPICs*, vol. 263, pp. 23:1–23:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). <https://doi.org/10.4230/LIPICs.ECOOP.2023.23>
 22. Olivieri, L., Tagliaferro, F., Arceri, V., Ruaro, M., Negrini, L., Cortesi, A., Ferrara, P., Spoto, F., Talin, E.: Ensuring determinism in blockchain software with golisa: an industrial experience report. In: Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. p. 23–29. SOAP 2022, Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3520313.3534658>
 23. Pnueli, M., Sharir, M.: Two approaches to interprocedural data flow analysis. *Program flow analysis: theory and applications* pp. 189–234 (1981)
 24. Zanatta, G., Caiazza, G., Ferrara, P., Negrini, L.: Inference of access policies through static analysis. *Int. J. Softw. Tools Technol. Transf.* **26**(6), 797–821 (2024). <https://doi.org/10.1007/S10009-024-00777-8>
 25. Zanatta, G., Caiazza, G., Ferrara, P., Negrini, L., White, R.: Automating ROS2 security policies extraction through static analysis. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2024, Abu Dhabi, United Arab Emirates, October 14–18, 2024. pp. 3627–3634. IEEE (2024). <https://doi.org/10.1109/IROS58592.2024.10802507>
 26. Zanatta, G., Ferrara, P., Lisovenko, T., Negrini, L., Caiazza, G., White, R.: Sound static analysis for microservices: Utopia? A preliminary experience with lisa. In: Stefano, L.D. (ed.) Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2024, Vienna, Austria, 20 September 2024. pp. 5–10. ACM (2024). <https://doi.org/10.1145/3678721.3686229>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Mopsa-C: Towards Incorrectness and Termination Verdicts (Competition Contribution)

Marco Milanese¹, Raphaël Monat²✉^{*}, Abdelraouf Ouadjaout¹, and
Antoine Miné¹

¹ LIP6, Sorbonne Université, F-75005, Paris, France

² Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRISTAL, F-59000 Lille, France

Abstract. We present advances we brought to Mopsa for SV-COMP 2026. Mopsa now supports a backward analysis mode which computes an under-approximation of the weakest liberal precondition for the alarms found during a verification pass, which generates an input harness to reproduce the incorrectness condition. We also added support for termination checking through the introduction of loop counters. With these improvements, Mopsa wins the *SoftwareSystems* category, and ranks third in the *TrueOverall* category.

Keywords: Static Analysis · Abstract Interpretation · Competition on Software Verification · SV-COMP.

1 Verification Approach : the Mopsa Platform

Mopsa is an open-source static analysis platform relying on abstract interpretation [6]. The implementation of Mopsa aims at exploring new perspectives for the design of static analyzers. Journault et al. [10] describe the core of Mopsa principles, and Monat [19, Chapter 3] provides an in-depth introduction to Mopsa’s design. The C analysis which we rely on for this competition is based on the work of Ouadjaout and Miné [27]; it proceeds by induction on the syntax, is fully context- and flow-sensitive. This is the fourth time Mopsa participates to SV-COMP [22, 20, 23]. In addition to a sound forward analysis, Mopsa now features a backward under-approximating analysis, and a sound termination analysis.

Backward Analysis. Mopsa now supports a backward analysis based on the abstract interpretation framework presented of Miné [17], Milanese and Miné [14, 15], which computes a liberal precondition. This precondition is used to infer counterexamples for alarms identified during the standard forward over-approximating analysis. The current implementation is experimental and does not support all the abstractions available in the analyzer (e.g., only the interval numeric abstraction is supported, while the forward analysis also supports polyhedra, string length, symbolic rewriting, etc). Nevertheless, the backward analysis supports all primary C constructs, including dynamic memory allocation

* Jury member

(using recency abstraction), low-level memory manipulation through pointers and casts (using a dynamically-typed cell-based memory abstraction), advanced control structures (goto and switch). If enabled, the backward analysis is automatically initiated at the end of the forward analysis whenever alarms are detected. It begins at the end of the program with a postcondition of **false**, and at any program location where a property violation is possible, it combines the current state with those states that *definitely* violate the property. As a result, the analysis ultimately produces a precondition that ensures either divergence or the violation of at least one alarm. Moreover, program inputs (e.g., `__VERIFIER_nondet`) are modeled as program variables, and in general the precondition found involves some constraints on the inputs.

Termination Analysis. We have implemented a forward termination analysis introducing loop counters and checking they are always bounded, following the previous works of Halbwachs [7], Saan et al. [31]. Additionally, we have extended the approach to handle backward gotos.

Other Improvements. We have implemented a preprocessing step ensuring boolean conditions (e.g., within conditionals and loops) are normalized, improving the precision of the analysis. We have refined the iterator handling backward gotos to minimize the number of statements used in the fixpoint iteration. Thanks to the SV-Benchmarks verdicts, we have also been able to fix a stub contract for the `exp` function of the C standard library, and to remove spurious overflow alarms which occurred when using variable length arrays.

2 Software Architecture: the SV-COMP Driver

By default, the C analysis of Mopsa detects all the runtime errors that may happen in the analyzed program, while SV-COMP tasks focus on a specific property at a time. We thus rely on an SV-COMP specific driver. It takes as input the task description (program, property, data model). It uses a sequential portfolio approach, by trying increasingly precise C analyses defined in Mopsa, possibly enabling also the backward version, until the property of interest is (dis)proved, or the most precise analysis is reached (or the resources are exhausted). Each analysis result is postprocessed by the driver to check if the property is proved. An analysis configuration defines the set of domains used and their parameters, allowing control of the precision-efficiency ratio.

Our portfolio is mostly unchanged from last year [23]. We streamlined the unrolling parameters used by default to improve scalability and rely on the autosuggestion hook to detect semantic patterns [23]. We are now using PPLite’s implementation of polyhedra [4], as the overall results seemed better than when using the default implementation of Apron [9].

Violation Verification The analyzer computes a liberal precondition, and to ensure that a value extracted from it constitutes a valid counterexample, it is necessary to verify that the program does not diverge—i.e., gets stuck in an infinite loop without violating the property. To this end, the driver first determines

a concrete input vector by assigning to each input variable the smallest value within its interval of variation. It then substitutes the input functions with these concrete values and compiles the benchmark. Specifically, to detect property violations such as overflows that may not result in program crashes, compilation is performed with C sanitizers enabled, as is done in [29]. After compilation, the program is executed. Whenever a crash is observed, the driver records the crash location and generates a corresponding violation witness. Notably, the crash is deterministic because all input calls are replaced with fixed, concrete values. Additionally, the program notifies the driver each time an input is accessed, enabling the inclusion of such inputs in the final witness. This information is used to determine both the order and the number of times each input function is invoked within the violation trace.

3 Strengths and Weaknesses

Mopsa participated in the following categories, targeting C programs: *ReachSafety*, *MemSafety*, *NoOverflows*, *SoftwareSystems* and *Termination*. An overview of results can be found in the competition report [5]. Mopsa is the best verifier of the *SoftwareSystems* track, which focuses on verifying real software systems. Overall, it also ranks third in the *TrueOverall* category where verifiers only focus on proving programs correct. We show in Figure 1 selected improvements of Mopsa as a whole. Figure 2 summarizes the result of the backward analysis, where we report both validated results and non-validated ones.

Strengths. Compared to last year’s participation Mopsa can now handle also incorrect programs. Thanks to this Mopsa managed to prove 2660 new tasks with false verdict that previously were not accessible due to the analysis method. Additionally, the support for proving program termination allowed Mopsa to prove 1010 tasks to be terminating.

Weaknesses. The implementation of the backward analysis is still experimental and not yet on par with the forward analysis. This is visible in Table 2 in *ReachSafety* where complex properties must be inferred—compared to simpler overflow or memory unsafe accesses—and *SoftwareSystems* where both scalability and precision on extended program sections are necessary. As a future work we would like to add support for more abstractions, as well as further tune the performance-precision tradeoff of this analysis.

Mopsa cannot soundly analyze concurrency-related verification tasks, but we could leverage previous abstract interpretation work targeting those properties [18, 34, 21, 32]. Our termination checker is quite naive and does not support recursive functions. We could reuse more involved—possibly backward—abstract interpretation techniques to prove termination [33, 24].

Our SV-COMP driver currently tries a sequence of increasingly precise configurations: this approach is not efficient, we are planning to develop techniques deciding what would be the best configuration to analyze a given program, following the works of Oh et al. [26], Mansur et al. [12], Wang et al. [35].

Mopsa is currently not generating meaningful correctness witnesses. Given that the witness validation limit has been significantly reduced this year, this is

Subcategory	Prop.	tasks	Mopsa'25	Mopsa'26	Best score, verifier (2026)	
Loops	R	764	386	491	982	aise [11]
Recursive	R	174	60	85	150	Symbiotic [1]
Sequentialized	R	585	4	18	531	CPAchecker [2]
Arrays	M	221	124	179	356	UAutomizer [8]
Juliet	M	3271	2530	3788	4709	CPAchecker [2]
Main	N	1986	2138	2384	2763	UParalizer [3]
BitVectors	T	34	—	10	55	PROTON [25]
ControlFlow	T	281	—	108	441	PROTON [25]
Heap	T	201	—	258	352	PROTON [25]
Other	T	1630	—	1576	2184	PROTON [25]
Busybox	N	65	8	14	14	Mopsa
Other	R	70	12	22	35	ESBMC-kind [13]
Other	M	49	14	18	18	Mopsa
DDL	T	215	—	32	201	UAutomizer [8]
Uthash	T	32	—	36	54	PROTON [25]

Fig. 1. Mopsa’s improvements for selected subcategories from ReachSafety, MemSafety, NoOverflows, Termination and SoftwareSystems track, where we compare the scores reached at SV-COMP 2025 and 2026. Property is either *ReachSafety*, *MemSafety*, *NoOverflow* or *Termination* (previously unsupported). The last columns show the score of Mopsa submitted last year, this year, and the best score reached by a verifier.

Category	false tasks	Validated	Unconfirmed	Best false , verifier	
ReachSafety	3076	137	36	1743	CPAchecker [2]
NoOverflows	3680	1473	92	2847	Symbiotic [1]
MemSafety	2156	1035	7	2042	Symbiotic [1]
SoftwareSystems	1568	15	4	324	Symbiotic [1]

Fig. 2. Mopsa’s backward analysis results for supported categories: number of tasks with a false verdict that Mopsa has been able to produce and validate, or to produce but that was unconfirmed by witness validators.

now a limitation of our approach, encountered for 27 of 9975 tasks where Mopsa finds the task to have result true. We have also noticed that some incorrectness witnesses are not validated. This is surprising given that our witnesses are fully concrete testcases. In the future, we plan to add a sanitizers-based validator, similar to SV-Sanitizer [30]—already used in our verifier—to confirm these tasks.

4 Software Project and Contributors

Mopsa is available on Gitlab [28], and released under an GNU LGPL v3 license. Mopsa was originally developed at LIP6, Sorbonne Université following an ERC Consolidator Grant award to Antoine Miné. Mopsa is now additionally developed in other places, including Inria, ENS, Airbus and Nomadic Labs. The people who improved Mopsa for SV-COMP 2026 are the authors of this paper.

Data-Availability Statement. The exact version of Mopsa and driver that participated to SV-COMP 2026 are available as a Zenodo archive [16].

Funding Statement. This work was supported by France’s Agence Nationale de la Recherche (ANR), program France 2030, reference ANR-22-PTCC-0001. This work was supported by project ANR-22-PECY0005 "Secureval" managed by the French National Research Agency for France 2030. This work was supported by grant agreement ANR-24-CE25-7956-01 RAISIN, and by an Amazon Research Award, Fall 2024.

Acknowledgments. We acknowledge the positive impact of Dagstuhl Seminar #25421 on this work.

Bibliography

- [1] Ayaziová, P., Jonáš, M., Mihalkovič, V., Sedláček, J., Strejček, J.: SYMBIOTIC 11: Predicate abstraction joins the party (competition contribution). In: Proc. TACAS (2), LNCS 16506, Springer (2026)
- [2] Baier, D., Beyer, D., Chien, P., Jankola, M., Kettl, M., Lee, N., Lemberger, T., Rosenfeld, M.L., Spiessl, M., Wachowitz, H., Wendler, P.: Cpatchecker 2.3 with strategy selection - (competition contribution). In: TACAS (3), Lecture Notes in Computer Science, vol. 14572, pp. 359–364, Springer (2024)
- [3] Barth, M., Dietsch, D., Heizmann, M., Jakobs, M.C.: ULTIMATE PARALIZER: Parallel trace abstraction (competition contribution). In: Proc. TACAS (2), LNCS 16506, Springer (2026)
- [4] Becchi, A., Zaffanella, E.: Pplite: Zero-overhead encoding of NNC polyhedra. *Inf. Comput.* **275**, 104620 (2020)
- [5] Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In: Proc. TACAS (2), LNCS 16506, Springer (2026)
- [6] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 238–252, ACM Press, Los Angeles, California (1977), <https://doi.org/10.1145/512950.512973>
- [7] Halbwachs, N.: Détermination automatique de relations linéaires vérifiées par les variables d’un programme. Theses, Institut National Polytechnique de Grenoble - INPG ; Université Joseph-Fourier - Grenoble I (Mar 1979), URL <https://theses.hal.science/tel-00288805>, universités : Université scientifique et médicale de Grenoble et Institut national polytechnique de Grenoble
- [8] Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüßle, F., Podelski, A.: ULTIMATE AUTOMIZER and the abstraction of

- bitwise operations (competition contribution). In: Proc. TACAS (3), pp. 418–423, LNCS 14572, Springer (2024), https://doi.org/10.1007/978-3-031-57256-2_31
- [9] Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV, pp. 661–667, Springer (2009)
- [10] Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: VSTTE, pp. 1–18 (2019), https://doi.org/10.1007/978-3-030-41600-3_1
- [11] Lin, Y., Chen, Z., Wang, J.: AISE v2.0: Combining loop transformations (competition contribution). In: Proc. TACAS (3), pp. 199–204, LNCS 15698, Springer (2025), https://doi.org/10.1007/978-3-031-90660-2_12
- [12] Mansur, M.N., Mariano, B., Christakis, M., Navas, J.A., Wüstholtz, V.: Automatically tailoring abstract interpretation to custom usage scenarios. In: CAV (2), Lecture Notes in Computer Science, vol. 12760, pp. 777–800, Springer (2021)
- [13] Menezes, R., Aldughaim, M., Farias, B., Li, X., Manino, E., Shmarov, F., Song, K., Brauße, F., Gadelha, M.R., Tihanyi, N., Korovin, K., Cordeiro, L.: ESBMC v7.4: Harnessing the power of intervals (competition contribution). In: Proc. TACAS (3), pp. 376–380, LNCS 14572, Springer (2024), https://doi.org/10.1007/978-3-031-57256-2_24
- [14] Milanese, M., Miné, A.: Generation of Violation Witnesses by Under-Approximating Abstract Interpretation. In: VMCAI, Springer (2024)
- [15] Milanese, M., Miné, A.: Under-approximating memory abstractions. In: Proc. of the 31th International Static Analysis Symposium (SAS’24), Lecture Notes in Computer Science (LNCS), vol. 14995, Springer (Oct 2024), <http://www-apr.lip6.fr/~mine/publi/article-milanese-al-sas24.pdf>
- [16] Milanese, M., Monat, R., Ouadjaout, A., Miné, A.: Mopsa at sv-comp 2026 (Nov 2025), <https://doi.org/10.5281/zenodo.17696794>
- [17] Miné, A.: Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. *Sci. Comput. Program.* **93**, 154–182 (2014), <https://doi.org/10.1016/J.SCICO.2013.09.014>, URL <https://doi.org/10.1016/j.scico.2013.09.014>
- [18] Miné, A.: Relational thread-modular static value analysis by abstract interpretation. In: VMCAI, Lecture Notes in Computer Science, vol. 8318, pp. 39–58, Springer (2014)
- [19] Monat, R.: Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries. Ph.D. thesis, Sorbonne Université, France (2021)
- [20] Monat, R., Milanese, M., Parolini, F., Boillot, J., Ouadjaout, A., Miné, A.: MOPSA-C: Improved verification for C programs, simple validation of correctness witnesses (competition contribution). In: Proc. TACAS (3), pp. 387–392, LNCS 14572, Springer (2024), https://doi.org/10.1007/978-3-031-57256-2_26

- [21] Monat, R., Miné, A.: Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. In: Bouajjani, A., Monniaux, D. (eds.) *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017*, Paris, France, January 15-17, 2017, *Proceedings, Lecture Notes in Computer Science*, vol. 10145, pp. 386-404, Springer (2017), https://doi.org/10.1007/978-3-319-52234-0_21, URL https://doi.org/10.1007/978-3-319-52234-0_21
- [22] Monat, R., Ouadjaout, A., Miné, A.: MOPSA-C: Modular domains and relational abstract interpretation for C programs (competition contribution). In: *Proc. TACAS (2)*, pp. 565-570, LNCS 13994, Springer (2023), https://doi.org/10.1007/978-3-031-30820-8_37
- [23] Monat, R., Ouadjaout, A., Miné, A.: MOPSA-C with trace partitioning and autosuggestions (competition contribution). In: *Proc. TACAS (3)*, pp. 229-235, LNCS 15698, Springer (2025), https://doi.org/10.1007/978-3-031-90660-2_17
- [24] Moussaoui Remil, N., Urban, C.: Termination Resilience Static Analysis. In: *VMCAI 2026 - 27th International Conference on Verification, Model Checking, and Abstract Interpretation*, Rennes, France (Jan 2026), URL <https://inria.hal.science/hal-05398150>
- [25] Mukhopadhyay, D., Metta, R., Karmarkar, H., Madhukar, K.: PROTON 2.1: Synthesizing ranking functions via fine-tuned locally hosted LLM (competition contribution). In: *Proc. TACAS (3)*, pp. 242-247, LNCS 15698, Springer (2025), https://doi.org/10.1007/978-3-031-90660-2_19
- [26] Oh, H., Lee, W., Heo, K., Yang, H., Yi, K.: Selective context-sensitivity guided by impact pre-analysis. In: *PLDI*, pp. 475-484, ACM (2014)
- [27] Ouadjaout, A., Miné, A.: A library modeling language for the static analysis of C programs. In: *SAS*, pp. 223-247 (2020), https://doi.org/10.1007/978-3-030-65474-0_11
- [28] Ouadjaout, A., Monat, R., Miné, A., Journault, M.: Mopsa (2022), URL <https://gitlab.com/mopsa/mopsa-analyzer>
- [29] Saan, S.: Sv-sanitizers in sv-comp 2026 (Oct 2025), <https://doi.org/10.5281/zenodo.17400380>, URL <https://doi.org/10.5281/zenodo.17400380>
- [30] Saan, S.: Sv-sanitizers: Sv-comp wrapper for sanitizers (Dec 2025), URL <https://github.com/sim642/sv-sanitizers>
- [31] Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: Goblint: Abstract interpretation for memory safety and termination - (competition contribution). In: Finkbeiner, B., Kovács, L. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024*, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, *Proceedings, Part III*, *Lecture Notes in Computer Science*, vol. 14572, pp. 381-386, Springer

- (2024), https://doi.org/10.1007/978-3-031-57256-2_25, URL https://doi.org/10.1007/978-3-031-57256-2_25
- [32] Suzanne, T., Miné, A.: Relational thread-modular abstract interpretation under relaxed memory models. In: Ryu, S. (ed.) *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings, Lecture Notes in Computer Science*, vol. 11275, pp. 109–128, Springer (2018), https://doi.org/10.1007/978-3-030-02768-1_6, URL https://doi.org/10.1007/978-3-030-02768-1_6
- [33] Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: Müller-Olm, M., Seidl, H. (eds.) *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings, Lecture Notes in Computer Science*, vol. 8723, pp. 302–318, Springer (2014), https://doi.org/10.1007/978-3-319-10936-7_19, URL https://doi.org/10.1007/978-3-319-10936-7_19
- [34] Vojdani, V., Apinis, K., Rötov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: the goblint approach. In: *ASE*, pp. 391–402, ACM (2016)
- [35] Wang, Z., Yang, L., Chen, M., Bu, Y., Li, Z., Wang, Q., Qin, S., Yi, X., Yin, J.: Parf: Adaptive parameter refining for abstract interpretation. In: *ASE*, pp. 1082–1093 (2024)

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Re3ver: Reverse and Verify (Competition Contribution)

Adéla Štěpková^{}*, Martin Jonáš^{}, and Jan Strejček^{}

Masaryk University, Brno, Czech Republic
{stepkova,martin.jonas,strejcek}@mail.muni.cz

Abstract. Most verifiers analyze a given program in a forward manner, i.e., along its control flow. However, there are many programs where the reason for the unreachability of error locations is near these locations. To efficiently verify these programs with forward verification algorithms, we designed and implemented a technique that reverses the given program before its verification. This paper describes a new experimental tool RE3VER, which combines the program reversal and the established verifier SYMBIOTIC based on program slicing and symbolic execution.

1 Verification Approach

Most state-of-the-art techniques for reachability traverse a given program in the forward direction—they start in the initial location and try to either find a feasible path to an error location or prove that no error location is reachable. However, there are many programs where the reason for the unreachability of error locations is closely before them. In such cases, it can be more efficient to perform the reachability analysis in the backward direction: from the error locations backwards to the initial locations. To prove that a program is correct, it suffices to show that the initial location cannot be reached from any error location in the backward direction. At the same time, a program is incorrect if the initial location is backward-reachable from some error location.

Several verification techniques perform some form of backward analysis, for instance *backward symbolic execution* [4,3] in software verification, or *reverse PDR* [11] and *complementary approximate reachability (CAR)* [9] in hardware model checking. These methods are mostly modifications or extensions of established forward techniques, and are usually able to solve some problem instances that the corresponding forward techniques cannot handle. Instead of modifying and reimplementing forward methods for backward analysis, we propose to reverse the input program itself. Intuitively, the reversed program starts its execution in the error locations of the original program, encodes all the possible paths against the original control flow, and turns the original initial location into an error location. This allows a wide range of existing forward verification techniques to essentially compute backward analysis of the original program when executed on the reversed program.

* Jury member

```

1  uint a = nondet();  1  uint a = nondet();
2  while (a > 0)        2  assume(a != 0);    // failed assert
3    a = a - 1;        3  assume(a <= 0);    // loop exited
4  assert(a == 0);     4  while (nondet()) { // loop entered?
                        5    a = a + 1;    // loop body
                        6    assume(a > 0); // loop not exited
                        7  }
                        8  assert(false); // start of P

```

Listing 1: Program P (left) and its reversed version $rev(P)$ (right)

Our technique transforms a given program P into a corresponding reversed program $rev(P)$. Roughly speaking, we reverse the direction of control flow between basic blocks, and we also reverse the content of each basic block, statement by statement. Some statements corresponding to injective functions can be directly transformed into statements with the opposite semantics (e.g., `i++` can be transformed into `i--`). All other statements are reversed with the use of nondeterminism and assumptions (e.g., `a = b * c;` can be reversed into the sequence `assume(a == b * c); a = nondet();`). For a full description of the program reversing algorithm, we refer to the Master thesis of Adéla Štěpková [13].

The process is demonstrated on the example given in Listing 1. The construction of the reversed program $rev(P)$ for the given program P proceeds as follows. The reversed program starts where the assertion of P fails: the variable `a` is initialized with a nondeterministic value that satisfies `a != 0`. Because the loop condition must not hold to exit the `while` loop in P , we assume its negation when entering it in $rev(P)$. As the number of iterations of the `while` loop in P is not known, $rev(P)$ calls `nondet()` in every iteration to guess if the loop should be exited or not. The statement `a = a - 1` in the loop body can be reversed into the statement `a = a + 1`. Because the loop condition must hold before every loop iteration in P , the reversed program assumes it holds after every reversed iteration. The call to `assert(false)` at the end of $rev(P)$ corresponds to the initial location of P . There is an execution of $rev(P)$ with a failing assertion if and only if there is an execution of P with a failing assertion.

The assertion in the original program P holds because of the loop condition and the fact that `a` is unsigned. However, some techniques like forward symbolic execution [7] would struggle to prove the assertion validity because of the enormous number of paths leading to the assertion (the paths differ in the number of loop iterations). On the contrary, basically every technique can easily prove the validity of the assertion in the reversed program: the assertion cannot fail because it is unreachable. Indeed, the first three lines of $rev(P)$ are already infeasible as there is no unsigned integer `a` satisfying both `a != 0` and `a <= 0`.

2 Software Architecture

The presented verification approach is implemented in a tool called RE3VER. The tool has been designed not to maximize the score achieved in the compe-

tion, but to show the potential of program reversing. The tool consists of two separate parts: REVERSER and SYMBIOTIC. REVERSER is a standalone implementation of the program reversing algorithm. It is written in C++ using the LLVM [8] library and it consumes and produces programs in LLVM IR as they have substantially less rich syntax than C programs. All the remaining functionality of RE3VER is provided by the verification framework SYMBIOTIC [5]. More precisely, SYMBIOTIC first compiles a given C program into LLVM IR using CLANG. REVERSER is then called to reverse it. If the reversing process fails, RE3VER aborts. In the opposite case, SYMBIOTIC performs program slicing (the code not relevant for reaching error locations is removed) and passes the sliced program to the symbolic executor JETKLEE, a fork of KLEE [2] integrated in SYMBIOTIC.

The last step is the construction of the corresponding verification witness. Unfortunately, SYMBIOTIC does not generate valid witnesses for the original programs because it is fed the reversed program. Thus, RE3VER outputs trivial witnesses that contain only the identification of the reachable error location in the witnesses of violated reachability safety. RE3VER identifies the first error location found in the input C program, which might not be the reached one in case of multiple error locations. This temporary solution will be fixed soon.

All components of RE3VER use LLVM 14. SYMBIOTIC is used in the version prepared for SV-COMP 2025, but configured differently: only the standard symbolic execution in JETKLEE is used, instead of a portfolio of techniques.

3 Strengths and Weaknesses

The main advantage of our approach is illustrated by the example in Listing 1: program reversing enables forward verification techniques, such as symbolic execution, to verify certain correct programs that it is unable to solve on its own.

We also emphasize that REVERSER can be combined with an arbitrary verifier that accepts LLVM IR programs on input. We chose SYMBIOTIC because of its solid performance. In principle, one could translate the reversed LLVM IR programs to C using LLVM2C [10] and apply an arbitrary verifier of C programs.

The main principal disadvantage of our approach is that the reversed programs are often significantly more complex than the original ones, and they heavily employ nondeterminism and assumptions. We presume that nondeterministic branching is currently the main obstacle for symbolic execution of reversed programs. Simplification of reversed programs is a topic for future work.

RE3VER also suffers from several other shortcomings caused by its early stages of development. First, RE3VER supports only the *unreach-call* property and participated only in *C.ReachSafety*. Further, REVERSER does not support several important features of C, such as structs, arrays, and pointers. Hence, it successfully reversed only 9427 out of 14390 benchmarks of *C.ReachSafety*.

Table 1 provides a quantitative comparison of the verification results of RE3VER and SYMBIOTIC in SV-COMP 2026 restricted to the 9427 reversed benchmarks. RE3VER produced 2 incorrect *true* results (excluded from Table 1, to be analyzed) and verified 138 benchmarks not decided by SYMBIOTIC. Recall

Table 1: The numbers of *true/false* benchmarks in base categories of *C.ReachSafety* reversed by REVERSER: the *total* number of benchmarks, solved by SYMBIOTIC, by RE3VER, and solved only by RE3VER (not by SYMBIOTIC).

Base category	Total		SYMBIOTIC		RE3VER		only RE3VER	
	true	false	true	false	true	false	true	false
Arrays	0	0	0	0	0	0	0	0
BitVectors	34	13	25	11	13	11	0	0
Combinations	154	336	0	193	0	26	0	0
ControlFlow	27	2	20	2	20	2	4	0
ECA	773	461	0	278	88	4	88	0
Floats	16	0	13	0	8	0	0	0
Hardness	5510	0	1081	0	348	0	4	0
Hardware	610	391	48	50	69	23	27	0
Heap	4	2	4	2	4	2	0	0
Loops	497	178	330	132	73	22	15	0
ProductLines	0	0	0	0	0	0	0	0
Recursive	52	59	37	45	6	30	0	0
Sequentialized	57	132	7	96	0	3	0	0
XCSP	60	59	50	41	14	8	0	0
Total	7794	1633	1615	850	643	131	138	0

that we compare against a SYMBIOTIC version that applies not only symbolic execution, but also *compact symbolic execution* [12], *backward symbolic execution with loop folding* [3], and *symbolic execution with predicate abstraction* [6].

4 Tool Setup, Usage, and Contributors

The binary of RE3VER is available at Zenodo [14]. The source code of REVERSER is available at <https://gitlab.fi.muni.cz/xstepkov/reverser>, the source code of SYMBIOTIC modified for RE3VER can be accessed at <https://github.com/adastepkova/symbiotic/tree/reversing>, both under the tag `svcomp26`. RE3VER can be run as

```
bin/symbiotic --sv-comp [--prp <prpfile>] [--32] --reverse <source>
```

where `<prpfile>` is a property file and `<source>` is the source C file. The only accepted property file is `unreach-call.prp`.

To compile RE3VER from sources, build each component separately. Compilation instructions for both components can be found in the `README.md` files in their respective repositories. Before running the tool, add the location of the REVERSER executable to the `PATH` environment variable.

The tool has been developed at the Faculty of Informatics of Masaryk University by Adéla Štěpková under the supervision of Jan Strejček and Martin Jonáš. It is available under the MIT license, and all its components and libraries (LLVM, SYMBIOTIC and its components) have permissive open-source licenses.

Acknowledgments. The authors were supported by the Czech Science Foundation project No. GA23-06506S.

Data Availability Statement. The complete SV-COMP 2026 [1] data set, including verification tasks, results, witnesses, and reproduction scripts, is available on the competition website: <https://sv-comp.sosy-lab.org/2026/>. The competition version of RE3VER is archived on Zenodo [14].

References

1. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (Report on SV-COMP 2026). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
2. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) OSDI 2008. pp. 209–224. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
3. Chalupa, M., Strejček, J.: Backward symbolic execution with loop folding. In: Dragoi, C., Mukherjee, S., Namjoshi, K.S. (eds.) SAS 2021. LNCS, vol. 12913, pp. 49–76. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_3
4. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: a powerful approach to weakest preconditions. In: Hind, M., Diwan, A. (eds.) PLDI 2009. pp. 363–374. ACM (2009). <https://doi.org/10.1145/1542476.1542517>
5. Jonáš, M., Kumor, K., Novák, J., Sedláček, J., Trtík, M., Zaoral, L., Ayaziová, P., Strejček, J.: Symbiotic 10: Lazy memory initialization and compact symbolic execution (competition contribution). In: Finkbeiner, B., Kovács, L. (eds.) TACAS 2024. LNCS, vol. 14572, pp. 406–411. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_29
6. Jonáš, M., Strejček, J., Griggio, A.: Combining symbolic execution with predicate abstraction and CEGAR. In: Narodytska, N., Rümmer, P. (eds.) FMCAD 2024. pp. 271–280. IEEE (2024). https://doi.org/10.34727/2024/ISBN.978-3-85448-065-5_33
7. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). <https://doi.org/10.1145/360248.360252>
8. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO 2004. pp. 75–88. IEEE Computer Society (2004), <https://doi.org/10.1109/CGO.2004.1281665>
9. Li, J., Zhu, S., Zhang, Y., Pu, G., Vardi, M.Y.: Safety model checking with complementary approximations. In: Parameswaran, S. (ed.) ICCAD 2017. pp. 95–100. IEEE (2017). <https://doi.org/10.1109/ICCAD.2017.8203765>
10. Mihalkovič, V., Jonáš, M., Strejček, J.: LLVM2C: Translation of LLVM IR to C for program analysis (2026), submitted to FM 2026.
11. Seufert, T., Scholl, C.: Sequential verification using reverse PDR. In: Große, D., Drechsler, R. (eds.) MBMV 2017. pp. 79–90. Shaker Verlag (2017)
12. Slabý, J., Strejček, J., Trtík, M.: Compact symbolic execution. In: Hung, D.V., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 193–207. Springer (2013). https://doi.org/10.1007/978-3-319-02444-8_15
13. Štěpková, A.: Reversing Programs for Error Reachability Analysis. Master’s thesis, Faculty of Informatics, Masaryk University, Brno (2025)
14. Štěpková, A., Strejček, J., Jonáš, M.: Re3ver: Submission to SVCOMP 2026 (Nov 2025). <https://doi.org/10.5281/zenodo.17672678>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





REFUNCTION: Conditional Termination by Abstract Interpretation of Numerical C Programs (Competition Contribution)

Naïm Moussaoui Remil[✉] and Caterina Urban[✉]

Inria & ENS | PSL, Paris, France
{naim.moussaoui-remil,caterina.urban}@inria.fr

Abstract. ReFuncTion is a static analyzer designed for proving conditional termination (resilience), and functional program properties expressed in Computation Tree Logic (CTL). The tool automatically infers piecewise-defined ranking functions and sufficient preconditions by means of abstract interpretation.

1 Verification Approach

REFUNCTION is an abstract-interpretation-based static analyzer for the verification of numerical C programs. It is a fork of the analyzer FUNCTION [10,11], from which it inherits conditional termination analysis [11] and the verification of conditional Computation Tree Logic (CTL) properties [14]. REFUNCTION extends FUNCTION with a new parser, support for signed machine integer, and non-recursive interprocedural analysis. Besides these programming-features extensions, REFUNCTION provides two new analysis: one for conditional termination resilience [7] and one for computing minimal set of variables that need to be controlled (i.e., constrained) to satisfy a given CTL property of interest [8].

1.1 Backward (Termination) Analysis

REFUNCTION proves termination (and other properties) by inferring a piecewise-defined ranking function [4,9], i.e., a function mapping program states to a well-ordered set and whose value strictly decreases during program execution.

The analysis implements the abstract-interpretation-based framework of Urban et al. [11,12], which soundly approximates the best ranking-function semantics of Cousot and Cousot [3]. It is performed backwards. For termination, REFUNCTION starts from the final program control point with a ranking function defined as the constant function `zero`. At each control point, it computes a piecewise ranking function whose domain under-approximates the set of states from which termination is guaranteed, and whose values over-approximate the remaining number of execution steps. The domain obtained at the initial control point therefore provides a sufficient condition for program termination.

1.2 Decision Tree Abstract Domain

The decision tree abstract domain (denoted $\mathcal{T}$) is used in REFUNCTION to represent and compute piecewise-defined ranking functions.

An element $t \in \mathcal{T}$ is a piecewise-defined partial function represented by a full binary tree, where each node — denoted $\text{NODE}\{c\}: t_1, t_2$, with $t_1, t_2 \in \mathcal{T}$ — is labeled by a constraint $c \in \mathcal{C}$ over the program variables (where $\mathcal{C}$ is the set of allowed constraints, e.g., linear constraints), and each leaf — denoted $\text{LEAF}: f$ — is labeled by a function $f \in \mathcal{F}$ of the program variables (where $\mathcal{F}$ is the set of allowed functions, e.g., affine functions). Nodes recursively partition the space of possible values of the program variables: constraint labeling nodes are satisfied by the left subtrees of the nodes, while the right subtrees satisfy their negation. The leaves of the tree represent the value of the function within each partition. The partitioning is dynamic: during the analysis, partitions (resp. decision nodes and constraints) are split (resp. added) by tests, modified by variable assignments and joined (resp. removed) when merging control flows. In order to minimize the cost of the analysis, a widening limits the height of the decision trees and the number of maintained partitions. Figure 1 shows an example of a decision tree. The leaf with value $\perp_F$ explicitly represents the undefined partition of the (partial) function. Undefined functions can also be represented by the leaf value $\top_F$ in case of an irrecoverable loss of precision of the analysis [2]. The decision tree in Figure 1 represents a function with value 1 when $z < x$ (at most one step to termination), with value 3 when $z \geq x \wedge z + c < x$ (at most three steps to termination) and undefined otherwise (termination is not proved when $z + c \geq x$).

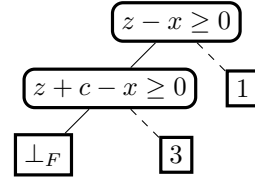


Fig. 1: Example of decision tree inferred by REFUNCTION.

1.3 Forward Reachability Analysis

REFUNCTION optionally performs a forward reachability analysis that computes an over-approximation of the reachable states at each program control point. When executed beforehand, this forward analysis restricts the state space by discarding unreachable states, thereby improving the precision analysis results.

1.4 New Features

REFUNCTION extends FUNCTION to analyze programs manipulating signed integers. This extension relies on the assumption that signed integer overflows do not occur. Consequently, signed machine integers are modeled as mathematical integers constrained to their respective type bounds.

While previous versions of REFUNCTION relied on function inlining, we have now implemented a context-insensitive interprocedural analysis. Operating

backwards, the analysis infers a decision tree t_1 . Upon encountering a function call, REFUNCTION analyzes the callee independently to compute an associated decision tree t_2 . The decision trees t_1 and t_2 are then joined by keeping paths defining the same sufficient preconditions and summing their associated values. This idea was originally proposed in [11] but was not implemented. In future versions, we plan to leverage function summaries to achieve a compositional analysis.

2 Software Architecture

REFUNCTION consists of 20 000 lines of OCaml code, and uses the parser of MOPSA [6] (available through its opam packages¹) to parse C programs. Only numerical programs, without bitwise operations, goto/break, unsigned machine integers and recursive functions are supported by our analyses. REFUNCTION returns a special error for any program using an unsupported feature, **true** when termination holds for any program state at the initial program control point, and **unknown** when for some initial program state the analysis is inconclusive. The analyzer does not generate witnesses.

The available abstract domains for the forward reachability analysis are those provided by the APRON library [5]. The decision tree abstract domain used for the backward termination analysis is implemented on top of the APRON library; the internal nodes of these trees correspond to univariate or multivariate linear constraints, and the leaves of the decision trees represent the value of the ranking functions as affine functions or ordinal-valued functions [13]. For the competition, REFUNCTION used convex polyhedra both for the reachability analysis and for the decision trees in the termination analysis.

3 Strengths and Weaknesses

Strengths. REFUNCTION is a *sound static analyzer*, therefore for the competition it produced no incorrect results [1]. We report 206 timeouts, 200 of which are due to the tasks in the ‘eca-rers2012’ benchmarks, which implement the same programming pattern. As the timeouts are caused by only one kind of program, it showcases the ability of REFUNCTION to converge and eventually produce results within the time limit. Besides, REFUNCTION produces results within reasonable time, with an average of CPU time of 2.7 s for the 111 correct results and 54 s for the 131 files for which it is inconclusive and returns **unknown**. Finally, we emphasize that the ability of REFUNCTION to find meaningful preconditions for program termination is an important feature, which unfortunately is not taken into account in the competition.

¹ <https://opam.ocaml.org/packages/mopsa/>

Weaknesses. Currently, REFUNCTION is unable to analyze most of the programs. It parses and reports an unsupported-feature error for 1926 tasks out of the 2384 tasks in the Termination category. REFUNCTION mainly suffers from the lack of sound abstract domains for non-numerical program features, e.g., heap-manipulating programs. The termination analysis returns **unknown** for 101 tasks for which termination holds. This is mostly due to a naïve widening operator [11] in the current implementation of the decision tree abstract domain. This could be addressed through state-of-the-art widening [2] or by developing new widening operators. Similarly, the absence of non-linear numerical abstract domains limits the precision of REFUNCTION.

4 Tool Setup, Configuration and Contributors

The analyzer code source and a Docker image are also available at our GitHub repository².

A README.md file describes the instructions to build the executable `main.exe` using the `dune` build system. A sample run command is:

```
./main.exe filename.c -domain polyhedra
```

REFUNCTION has only participated in the Termination category of SV-COMP 2026.

REFUNCTION is developed at École Normale Supérieure and Inria Paris by Naïm Moussaoui Remil under the supervision of Caterina Urban. We thank all the contributors of FUNCTION [10,11] from which we forked to develop REFUNCTION.

Data Availability. REFUNCTION’s competition version is distributed as a Zenodo archive³. The tool-info module and the benchmark definition files for SV-COMP 2026 are named `function-res`.

Acknowledgements. This work has benefitted from the participation of the authors in the Dagstuhl Seminar 25421 “Sound Static Program Analysis in Modern Software Engineering”.

References

1. D. Beyer and J. Strejček. Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In *Proc. TACAS (2)*, LNCS 16506. Springer, 2026.
2. Nathanaël Courant and Caterina Urban. Precise Widening Operators for Proving Termination by Abstract Interpretation. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 136–152, 2017.

² <https://github.com/naim-mr/refunction>

³ <https://zenodo.org/records/17525389>

3. Patrick Cousot and Radhia Cousot. An Abstract Interpretation Framework for Termination. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 245–258. ACM, 2012.
4. Robert W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.
5. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.
6. Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Mopsa-c with trace partitioning and autosuggestions (competition contribution). In Arie Gurfinkel and Marijn Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part III*, volume 15698 of *Lecture Notes in Computer Science*, pages 229–235. Springer, 2025.
7. Naïm Moussaoui Remil and Caterina Urban. Termination Resilience Static Analysis. In *VMCAI 2026 - 27th International Conference on Verification, Model Checking, and Abstract Interpretation*, Rennes, France, January 2026.
8. Naïm Moussaoui Remil, Caterina Urban, and Antoine Miné. Automatic detection of vulnerable variables for CTL properties of programs. In Nikolaj S. Bjørner, Marijn Heule, and Andrei Voronkov, editors, *LPAR 2024: Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning, Port Louis, Mauritius, May 26-31, 2024*, volume 100 of *EPiC Series in Computing*, pages 116–126. EasyChair, 2024.
9. Alan Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.
10. C. Urban. FUNCTION: An abstract domain functor for termination (competition contribution). In *Proc. TACAS*, LNCS 9035, pages 464–466. Springer, 2015.
11. Caterina Urban. *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs (Analyse Statique par Interprétation Abstraite de Propriétés Temporelles Fonctionnelles des Programmes)*. PhD thesis, École Normale Supérieure, Paris, France, 2015.
12. Caterina Urban and Antoine Miné. A Decision Tree Abstract Domain for Proving Conditional Termination. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, volume 8723 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2014.
13. Caterina Urban and Antoine Miné. An Abstract Domain to Infer Ordinal-Valued Ranking Functions. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 412–431. Springer, 2014.
14. Caterina Urban, Samuel Ueltschi, and Peter Müller. Abstract interpretation of CTL properties. In Andreas Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, pages 402–422. Springer, 2018.

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





SEAL: Symbolic Execution with Separation Logic (Competition Contribution)

Tomáš Brabec¹ , Tomáš Dacík ^{*}¹ , and Tomáš Vojnar^{1,2}

¹ Faculty of Information Technology, Brno University of Technology, Czech Republic

² Faculty of Informatics, Masaryk University, Czech Republic

Abstract. SEAL is a static analyser for the verification of programs that manipulate unbounded linked data structures. It is based on separation logic to represent abstract memory states and, unlike other separation-logic-based approaches, it employs a general-purpose separation logic solver ASTRAL for satisfiability and entailment checking, which itself is based on translation to SMT. This design results in a modular architecture intended to be easier to extend and to combine with reasoning in other theories. Although still a prototype, SEAL achieved competitive results in the *LinkedLists* base category and was one of only four analysers capable of verifying programs with unbounded lists. We believe that the tool’s extensibility, combined with further development, can lead to significant improvements in future competitions.

1 Verification Approach

SEAL is a shape analyser based on forward abstract interpretation over formulae of separation logic (SL) [19], inspired by the approach of [12]. Unlike the vast majority of SL-based analysers, SEAL relies on a general-purpose SL solver ASTRAL [10] to check satisfiability and entailment of SL formulae (instead of using some dedicated heuristic approach for this purpose). As a result, analysis and logical reasoning in SL are cleanly decoupled. The ASTRAL solver was originally developed by the authors of this paper and was further optimized and adapted for efficient handling of verification queries within this work. Furthermore, since the decision procedure implemented in ASTRAL is based on a translation of SL into SMT, SEAL can seamlessly integrate SL with reasoning in other theories.

As SEAL is a relatively new tool, its potential has not yet been fully exploited. Currently, our analyser supports only three built-in data structures and simple integer properties. In the rest of the paper, we describe the architecture of the tool and its current capabilities. More details can be found in [6] (where a slightly earlier version of SEAL is referred to as KTSN).

Abstract domain. SEAL represents abstract states using so-called *symbolic heaps*, i.e., SL formulae of the form $\exists x_1, \dots, x_m. \varphi_1 * \dots * \varphi_n$ where $*$ denotes a separating conjunction and each φ_i is an atomic formula. Atomic formulae are either *pure* describing (dis)equalities of pointers or constraints over non-pointer values, *points-to* atoms describing a single heap cell, or *inductive predicates* used to describe unbounded sequences of pointers of a particular shape. A special predicate

* Jury member.

is used to explicitly represent memory that was freed and is still reachable. The abstract domain can also represent precise values of integer variables within the range $[-k, k]$ where $k = 5$ by default. For each program location, a disjunction of symbolic heaps representing the reachable memory configurations is computed and represented as a set of symbolic heaps.

Currently, SEAL supports three kinds of data structures: singly-linked lists (SLLs), doubly-linked lists (DLLs), and nested singly-linked lists (NLLs). In all three cases, the structure is acyclic and possibly empty. Some cyclic structures can still be represented explicitly using multiple acyclic structures.

Abstraction. SEAL performs widening based on abstraction to generalise the shape of the memory by replacing its parts by instances of inductive predicates. The abstraction is performed by finding pairs of compatible spatial predicates (points-to or list predicates) and joining them into a single list predicate—provided that the memory location linking them is not the target of some external pointer links and it is not pointed to by a program variable. The minimum length of a list predicate is tracked up to a pre-defined, fixed limit.

Currently, SEAL guesses an expected inductive predicate for each C structure before the analysis. A structure with one or two self-referential fields is classified as an SLL or DLL, respectively. A structure with a single self-referential field and an additional field previously classified as an SLL is classified as an NLL. During the analysis, SEAL just checks whether such an abstraction can be performed.

The above approach can fail in cases like the following structure TSLL defined as `{struct TSLL* next; struct TSLL* inner;}` and intended to be used to build an NLL with the `next` pointer unused in the nested lists. For this structure, SEAL will guess its type to be DLL because it has two self-referential fields. This weakness could be improved by selecting the suitable inductive predicate for abstraction during the analysis. We would like to improve this aspect in the future together with adding more predicates for more data structures. We also want to explore methods for synthesising inductive predicates from examples.

Finding loop fixpoints. The analysis reaches a fixpoint for a loop header location when the entailment between the symbolic state S from the current loop iteration and the state T from the previous loop iteration is valid. The entailment is approximated by an incomplete method checking whether for each symbolic heap $S_i \in S$ there exists $T_j \in T$ such that $S_i \models T_j$. The pairs (S_i, T_j) to be checked are chosen according to a heuristic based on the numbers of allocated memory locations in S_i and T_j . For an entailment to be valid, there cannot be more allocations on the right side of the entailment than there are on the left side. The same method is used for pruning redundant symbolic heaps in the join operator. Additionally, a number of syntactic simplifications are applied during analysis, such as reduction of equivalence classes in formulae, removal of variables outside of their scope, syntactic deduplication of formulae, etc.

Integer variables. Currently, SEAL can only track concretely known values of integer variables that are within a pre-configured range. Conditions and arithmetic operations on these variables can be evaluated precisely. When the result of an operation on such values goes out of the represented range, the variable that

Table 1: The results of selected tools for *MemSafety-LinkedLists* and its subset with unbounded lists. The columns give numbers of correctly verified programs and found bugs, programs with no result reported (error, unknown, or out of resources), and the total time used for programs for which a result was produced.

Verifier	All programs (134)				Unbounded (69)			
	True	False	No res.	Time [s]	True	False	No res.	Time [s]
PREDATORHP	96	28	10	112	46	14	9	77
CPACHECKER	87	27	20	2158	40	13	16	1550
SEAL	55	4	75	571	14	2	53	166
2LS	46	14	73	1154	3	13	53	92

should receive the result becomes unconstrained instead. The range of tracked integer values can be configured; increasing it trades precision for analysis time.

Interprocedural analysis. Analysis of function calls is optimised by storing function summaries – pairs of formulae representing precondition and postcondition of the analysed function. Using this information, a consecutive call of a function may be replaced by applying a matching summary.

Analysis results. SEAL is able to verify absence of NULL-pointer dereferences and use-after-free errors (**valid-deref** property), double-free errors (**valid-free** property), and some memory leaks (those that fall under the **valid-memtrack** property). The **valid-memcleanup** property is not yet supported. When an issue is found after reaching a condition that could not be precisely evaluated based on the current state (e.g., because of missing integer information), the analyser returns the UNKNOWN result. However, false positives are still possible because of the abstraction. In the future, they could be avoided by validating the issue by rerunning the analyser without abstraction. When an unsupported language feature is reached, the ERROR result is returned.

SL solver. We briefly describe how the backend solver ASTRAL of SEAL handles SL queries. Given an SL formula, ASTRAL first computes a bound on how many times it suffices to unfold the predicate instances present in it and then uses this bound to translate the SL formula into several formulae in SMT theories of bitvectors and arrays. Those formulae are then passed to an SMT solver, which is configured to be BITWUZLA [17] for SV-COMP. Unlike in [10], ASTRAL runs in a mode supporting user-defined inductive predicates, i.e., the predicates are defined by SEAL and ASTRAL runs without any built-in support for them [11].

2 Strengths and Weaknesses

Table 1 provides a view on the results of *MemSafety-LinkedLists* [4], the only base category in which SEAL participates, illustrating the strengths of SEAL. Since almost half of the programs in this category manipulate lists of bounded size only and can therefore be verified using bounded model checking (or similar heuristics), we focus on the subset of benchmarks that require reasoning about unbounded data structures. Specifically, we exclude the benchmark

sets `list-simple` and `list-ext3-properties`, which operate exclusively on bounded lists, as well as `ddv-machzwd` where the list-manipulating code is unreachable. Only the tools that can verify correctness of at least one of the remaining programs are shown in the table.

Strengths. As Table 1 shows, out of 20 tools (excluding metaverifiers) participating in *MemSafety*, only 4 of them were able to verify correctness of programs with unbounded data structures. This indicates that reasoning about such programs remains a challenging task. SEAL can verify more programs than 2LS [14, 15], but less than PREDATORHP [13, 18] and CPACHECKER [1, 2], two analysers based on *symbolic memory graphs* (SMG). We believe that the biggest strength of SEAL lies in its potential to be extended for data structures that cannot be represented by SMGs, such as skip-lists or trees. Indeed, ASTRAL in its new versions supports user-defined inductive predicates, including various kinds of trees, which may be implemented in SEAL or potentially learned automatically. We would also like to use the generality of our technique to create a witness validator for correctness memory safety witnesses, once a witness format is established.

Weaknesses. As an early prototype, SEAL comes with a handful of limitations. First of all, we support only a subset of the C language (most kinds of pointer arithmetic, integer fields in list structures, global and static variables, arrays, and other language features are not supported). For this reason, we do not participate in other base categories of *MemSafety*. SEAL currently has limited support for non-pointer values, only tracking precise values of integer variables, where known. For this reason, we are often not able to report a memory safety violation because an unknown condition in the program was reached, making any violation result a possible false positive. This limitation can be eliminated by tracking more complex information in symbolic states or by embedding our analyser as a new abstract domain into the Frama-C’s abstract interpreter EVA [5].

3 Software Project and Tool Setup

SEAL is implemented in OCaml as a plugin of the Frama-C platform [3, 9] (version 31.0). Currently, it is a stand-alone plugin that does not communicate with other analysers of Frama-C. SEAL relies on the Frama-C’s intermediate representation (a fork of the CIL project [16]) and data flow analysis engine. Optionally, SEAL can visualise its detailed results using the Frama-C’s new GUI frontend Ivette [8]. We used the ASTRAL solver version 2.0-prerelease.1 that internally uses the BITWUZLA SMT solver version 0.8.2 through its OCaml binding.

Usage. A binary of SEAL is available at Zenodo [7] (the only runtime dependency is GCC). In SV-COMP, SEAL is run via a Python3 script `seal-entrypoint.py`:

```
./seal-entrypoint.py -machdep=[gcc_x86_32|gcc_x86_64] input.c
```

See the attached file `README.md` for more information.

Software project. SEAL is available under the MIT license, the project is maintained by Tomáš Brablec. The ASTRAL solver which was modified as a part of this work is maintained by Tomáš Dacík.

Data-Availability Statement. SEAL is available under the MIT license at <https://github.com/pepega007xd/seal>. The version participating in SV-COMP’26 is available under the tag `svcomp26` and archived at Zenodo [7].

Acknowledgements. We appreciate discussion with members of the Frama-C team at CEA, France. The research was supported by the Czech Science Foundation project 26-22640S. Tomáš Dacík was supported by the Brno PhD Talent Scholarship of the Brno City Municipality. The collaboration with CEA was supported by the project VASSAL: “Verification and Analysis for Safety and Security of Applications in Life” funded by the European Union under Horizon Europe WIDERA Coordination and support Action/Grant Agreement No. 101160022. 

References

1. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPAchecker 2.3 with Strategy Selection (Competition Contribution). In: Proc. of TACAS (3). pp. 359–364. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
2. Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Software Verification with CPAchecker 3.0: Tutorial and User Guide. In: Proc. of FM. pp. 543–570. LNCS 14934, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_30
3. Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., Williams, N.: The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform. Commun. ACM **64**(8), 56–68 (2021). <https://doi.org/10.1145/3470569>
4. Beyer, D., Strejček, J.: Evaluating Software Verifiers for C, Java, and SV-LIB (Report on SV-COMP 2026). In: Proc. of TACAS (2). LNCS 16506, Springer (2026)
5. Blazy, S., Bühler, D., Yakobowski, B.: Structuring Abstract Interpreters Through State and Value Abstractions. In: Proc. of VMCAI. pp. 112–130. LNTCS 10145, Springer (2017). https://doi.org/10.1007/978-3-319-52234-0_7
6. Brablec, T.: Static Analysis of Heap-Manipulating Programs using Separation Logic. Bachelor’s thesis, Faculty of Information Technology, Brno University of Technology, Brno (2025), <http://hdl.handle.net/11012/253189>
7. Brablec, T., Dacík, T., Vojnar, T.: SEAL (SV-COMP 2026). Zenodo (2025). <https://doi.org/10.5281/zenodo.17690232>
8. Correnson, L.: Ivette: A Modern GUI for Frama-C. In: Proc. of SEFM. pp. 116–131. LNCS 13765, Springer (2022). https://doi.org/10.1007/978-3-031-26236-4_10
9. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A Software Analysis Perspective. In: Proc. of SEFM. pp. 233–247. LNPSE 7504, Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16

10. Dacík, T., Rogalewicz, A., Vojnar, T., Zuleger, F.: Deciding Boolean Separation Logic via Small Models. In: Proc. of TACAS. pp. 188–206. LNCS 14570, Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_11
11. Dacík, T., Rogalewicz, A., Sighireanu, M., Vojnar, T., Zuleger, F.: Deciding Separation Logic with Inductive Definitions via Small Models and Translation to SMT (2026), under preparation for submission.
12. Distefano, D., O’Hearn, P.W., Yang, H.: A Local Shape Analysis Based on Separation Logic. In: Proc. of TACAS. pp. 287–302. LNCS 3920, Springer (2006). https://doi.org/10.1007/11691372_19
13. Dudka, K., Peringer, P., Vojnar, T.: Byte-Precise Verification of Low-Level List Manipulation. In: Proc. of SAS. pp. 215–237. LNPSE 7935, Springer (2013). https://doi.org/10.1007/978-3-642-38856-9_13
14. Malík, V., Hruška, M., Schrammel, P., Vojnar, T.: Template-Based Verification of Heap-Manipulating Programs. In: Proc. of FMCAD. p. 103–111. IEEE (2018). <https://doi.org/10.23919/FMCAD.2018.8603009>
15. Malík, V., Schrammel, P., Vojnar, T., Nečas, F.: 2LS: Arrays and Loop Unwinding (Competition Contribution). In: Proc. of TACAS (2). pp. 529–534. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_31
16. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Proc. of CC. pp. 213–228. LNCS 2304, Springer (2002). https://doi.org/10.1007/3-540-45937-5_16
17. Niemetz, A., Preiner, M.: Bitwuzla. In: Proc. of CAV. pp. 3–17. LNCS 13965, Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_1
18. Peringer, P., Šoková, V., Vojnar, T.: PredatorHP Revamped (Not Only) for Interval-Sized Memory Regions and Memory Reallocation (Competition Contribution). In: Proc. of TACAS (2). pp. 408–412. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_30
19. Reynolds, J.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proc. of LICS. pp. 55–74. IEEE (2002). <https://doi.org/10.1109/LICS.2002.1029817>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





SWAT: Improvements to the Symbolic Executor (Competition Contribution)

Nils Loose* (✉) , Florian Sieck , Felix Mächtle , and Thomas Eisenbarth 

Institute for IT Security, University of Luebeck, Luebeck, Germany
`{n.loose, f.sieck, f.maechtle, t.eisenbarth}@uni-luebeck.de`
<https://www.its.uni-luebeck.de/en/research/tools/swat>

Abstract. SWAT is an instrumentation-based dynamic symbolic execution framework for Java programs running on the standard Java Virtual Machine (JVM). The framework’s two main components are the symbolic executor that collects traces during concrete execution and a decoupled symbolic explorer to systematically explore the collected program paths. This paper describes recent improvements made to SWAT for the SV-COMP 2026 Java category including the first-time participation in the new runtime track. To explicitly reason about exceptional program flows, symbolic modelling of instructions that may throw exceptions is introduced into SWAT. Furthermore, various performance optimizations were implemented, including a migration from the theory of arithmetic to the theory of fixed-size bit-vectors for symbolic reasoning. Finally, 124 new symbolic models were added to significantly increase the coverage of supported Java Standard Library methods. Overall these improvements led to a significant increase in SWAT’s performance and capabilities, reaching a shared first place in the **valid-assert** subcategory and fourth place overall in the Java category.

Keywords: Dynamic Symbolic Execution · Java · Dynamic Instrumentation

1 Overview

SWAT is a dynamic symbolic execution engine (DSE) that is designed to analyse Java programs running on the standard Java Virtual Machine (JVM). Bytecode is instrumented with ASM [3] during classloading. The instructions added by the instrumentation dynamically build and maintain a shadow state of the execution. With the shadow state, SWAT records a trace of symbolic path constraints using JavaSMT [1]. A decoupled symbolic explorer solves these path constraints to guide exploration based on a dynamically assembled execution tree. An overview of the execution lifecycle is shown in Figure 1. A more detailed description of the architecture can be found in the original SV-COMP contribution paper [4]. This paper focuses on the improvements made for this year’s competition. In short, the main additions are:

* Jury member

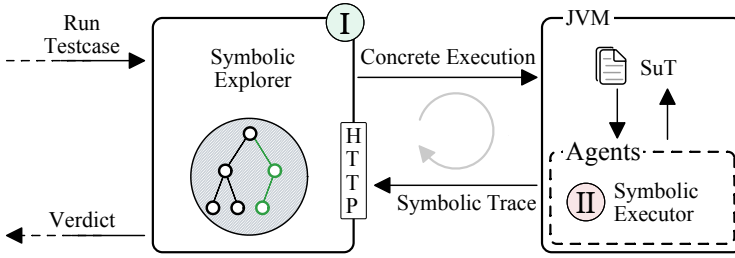


Fig. 1. Interaction between the symbolic explorer (I) and the symbolic executor (II) during verification task exploration. The explorer starts the verification task with concrete inputs and attaches the symbolic executor as agent. Additionally, it manages the execution tree that is annotated with symbolic path constraints returned from the executor. On startup of the system under analysis, the executor instruments the bytecode to extract symbolic information during concrete execution.

- **Runtime Exception Modelling:** Explicit symbolic modelling for instructions that may throw runtime exceptions to enable reasoning about exceptional program flows and support the new `runtime-exception` property.
- **Theorem Changes:** Migration of symbolic reasoning from the theory of arithmetic to the theory of fixed-size bit-vectors for `Double` and `Float` primitives to increase performance and accuracy.
- **New Symbolic Models:** Addition of 124 new symbolic models for Java Standard Library methods to increase the range of supported programs.

2 Improvements

Runtime Exception Modelling. This year, the property `runtime-exception` was moved from demo [2] into the main Java track alongside the established `assert-java` property. This property requires the verification tool to check whether a program can throw any uncaught runtime exception during its execution. To determine if a program violates this property, it is, in contrast to the assertion violation property, not sufficient to fully explore the program paths.

Consider the instruction `ILOAD` that loads an integer from an array. This instruction may throw an `ArrayIndexOutOfBoundsException` if the index is out of bounds. However, exceptional behaviour is data-dependent and may or may not occur on a dynamically explored path. An additional symbolic constraint is required to determine whether an index exists that causes the exception to be thrown. To symbolically reason about such exceptions during exploration, we handle each instruction that may throw a runtime exception as a branching instruction and, if possible, the condition to take the exceptional path is modelled symbolically.

This allows SWAT to explicitly explore exceptional program flows and determine whether an uncaught runtime exception can occur. Formally, for an array

access instruction with symbolic index i and symbolic array length ℓ , we generate the exceptional path condition:

$$\varphi_{exc} = (i < 0) \vee (i \geq \ell)$$

The negation $\neg\varphi_{exc}$ guards the normal execution path. This pattern generalizes to other potentially-throwing instructions such as `IDIV` which throws for divisions by zero. However, for instructions where the exceptional path depends on null checks or type compatibility (e.g., `CHECKCAST`), we cannot yet model these conditions symbolically.

Theorem Changes. The initial version of SWAT used in previous years relied on the unbounded theory of arithmetic for symbolic reasoning on all integral primitive types. However, this required explicit bound handling, offered limited support for bit-based operations, and complicated datatype translation. For SV-COMP 2026, we migrated the symbolic executor to the theory of fixed-size bit-vectors, which offers several advantages. First, it allows for native bounds handling and eases casting between floating-point and integer primitives. Additionally, bit-based operations are natively supported and overall, this theory offers better performance for symbolic reasoning.

New Symbolic Models. The instrumentation scope of SWAT is configurable and usually targets the namespace of the system under analysis. Choosing a narrow scope allows more efficient symbolic execution but requires symbolic models for invocations of uninstrumented methods. This commonly occurs for standard library methods, which require symbolic models to maintain symbolic tracking across method boundaries. In total, the Java track invokes 425 unique method signatures from the Java Standard Library. With this year’s addition of the `autostub` [7] verification tasks, new benchmarks that explicitly test a verifier’s capabilities on library methods were introduced. We added 124 new symbolic models covering the wrapper classes of primitive types and various utility classes.

3 Strengths and Weaknesses

SWAT participated in the *Java Overall* category, placing fourth overall and shared first with SPF [8] in the established *valid-assert* subcategory. Generally, SWAT performs better on violation cases and has produced no false positives across all categories. While the exploration strategy is currently a simple depth-first search that is not guided towards potential violation paths, the presence of a counterexample allows SWAT to quickly terminate exploration. Safe cases, on the other hand, require full exploration of all program paths as well as no loss of symbolic context or precision. Such loss can occur due to, e.g., missing or imprecise symbolic models. The executor detects such losses and as a precaution does not produce a safe verdict. Additionally, as is common for dynamic symbolic execution engines, unbounded loops or recursion quickly lead to path explosion and prevent full exploration of the state space.

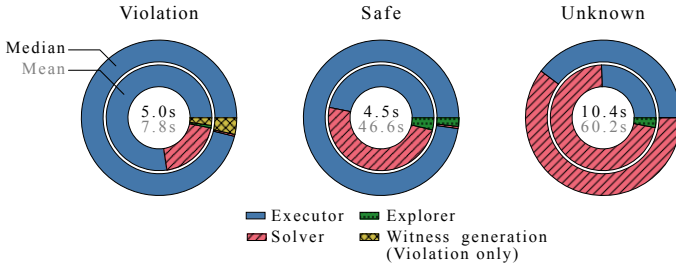


Fig. 2. Split of the total runtime per verification task across SWAT’s components for **assert-java** with at least one iteration of solving. The cases are distinguished by correctly identified violations (left), correctly identified safe cases (middle), and cases where SWAT fails to give a verdict (right).

Figure 2 shows the split of the total runtime per verification task across SWAT’s components for the **assert-java** property, distinguishing between correctly identified violations, correctly identified safe cases, and cases where no verdict was reached. For most cases, the executor dominates the runtime, as reflected in the median distributions. However, the mean distributions reveal that for more difficult cases, particularly those resulting in no verdict, the time spent in the solver increases significantly.

The current interaction between the explorer and the executor is not ideal for the SV-COMP use case, as the executor is intended to be started once, then kept alive during exploration. The current design targets scenarios such as web service testing, in which an iteration does not reflect a restart of the JVM but rather a new request to an already running system. In the SV-COMP setting, however, the executor is started fresh for each path exploration, leading to significant overhead due to repeated classloading and instrumentation. This is not a conceptual limitation of SWAT, and we intend to address this in future competition contributions by improving the explorer-executor interaction for the SV-COMP use case.

4 Software Project

SWAT is developed and maintained by the Institute for IT Security at the University of Luebeck. The source code is available on GitHub¹, and installation instructions, documentation, as well as examples can be found on our GitHub Pages site². SWAT offers a modular architecture with a variety of driver modules that dictate the behaviour of the symbolic executor and explorer. The specific configurations used for SV-COMP are set in the global configuration file³.

¹ <https://github.com/SWAT-project/SWAT/>

² <https://swat-project.github.io/docs/>

³ <https://github.com/SWAT-project/SWAT/blob/SV-Comp-Submission-2026/sv-comp.cfg>

Data Availability Statement. The version of SWAT used for the SV-COMP 2026 Java category is available at Zenodo [5] and on GitHub [6].

5 Acknowledgments

This work has been supported by the project *Vertrauenswürdiger Datenaustausch für verteilte Stromnetze (VeDS)* within the framework of the KI-Förderrichtlinie (AI funding guideline) by the State of Schleswig-Holstein.

References

1. Baier, D., Beyer, D., Friedberger, K.: Javasmmt 3: Interacting with SMT solvers in java. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 195–208. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_9, https://doi.org/10.1007/978-3-030-81688-9_9
2. Beyer, D., Strejcek, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Gurfinkel, A., Heule, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part III. Lecture Notes in Computer Science, vol. 15698, pp. 151–186. Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_9, https://doi.org/10.1007/978-3-031-90660-2_9
3. Bruneton, E., Lenglet, R., Coupaye, T.: Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* **30**(19) (2002)
4. Loose, N., Mächtle, F., Sieck, F., Eisenbarth, T.: SWAT: modular dynamic symbolic execution for java applications using dynamic instrumentation (competition contribution). In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III. Lecture Notes in Computer Science, vol. 14572, pp. 399–405. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_28, https://doi.org/10.1007/978-3-031-57256-2_28
5. Loose, N., Sieck, F., Mächtle, F., Eisenbarth, T.: Swat artifact for sv-comp 2026 (Nov 2025). <https://doi.org/10.5281/zenodo.17748741>, <https://doi.org/10.5281/zenodo.17748741>
6. Loose, N., Sieck, F., Mächtle, F., Eisenbarth, T.: SWAT Competition Version. <https://github.com/SWAT-project/SWAT/tree/SV-COMP-Submission-2026> (2025), accessed 12/2025
7. Mächtle, F., Loose, N., Serr, J., Sander, J., Eisenbarth, T.: Autostub: Genetic programming-based stub creation for symbolic execution. In: IEEE/ACM International Workshop on Search-Based and Fuzz Testing, SBFT 2025, Ottawa, ON, Canada, April 28-29, 2025. pp. 19–22. IEEE (2025). <https://doi.org/10.1109/SBFT66712.2025.00011>, <https://doi.org/10.1109/SBFT66712.2025.00011>

8. Noller, Y., Păsăreanu, C.S., Le, X.B.D., Visser, W., Fromherz, A.: Symbolic PATHFINDER for SV-COMP (competition contribution). In: Proc. TACAS (3). pp. 239–243. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_21

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Symbiotic 11: Predicate Abstraction Joins the Party (Competition Contribution)

Paulína Ayaziová^{ID}, Martin Jonáš^{ID*}, Vincent Mihalkovič^{ID},
Jindřich Sedláček^{ID}, and Jan Strejček^{ID}

Masaryk University, Brno, Czech Republic

Abstract. SYMBIOTIC 11 is the first version of SYMBIOTIC that features predicate abstraction. We implemented the recently introduced technique of symbolic execution with implicit predicate abstraction in the underlying symbolic executor SLOWBEAST. Additionally, we improved non-termination checking and witness production. We show that these changes have positive effect on the overall performance of SYMBIOTIC 11.

1 Verification Approach

SYMBIOTIC is based on program instrumentation, static slicing, and ultimately relies on several variants of symbolic execution. As the workhorse, SYMBIOTIC 11 uses the classic *symbolic execution* (SE) [12] and to improve proving power on safe programs, it also implements *backward symbolic execution with loop folding* (BSELF) [7], and *compact symbolic execution* (CSE) [14]. As a major addition this year, SYMBIOTIC 11 also implements recently proposed *symbolic execution with implicit predicate abstraction* (SEWPA) [11].

The main verification workflow is the same as in the previous versions and is explained in detail in previous system descriptions [6,10]. We briefly recall it in the next subsection and then focus on changes in SYMBIOTIC 11.

Verification workflow for safety properties For the property `unreach-call`, SYMBIOTIC first applies static slicing to prune the parts of the code that can be proven to be irrelevant for reaching the target function. It then executes a sequential portfolio of the symbolic execution techniques mentioned above.

For `no-overflow` property, SYMBIOTIC runs a range analysis that identifies instructions that can potentially cause a signed integer overflow. These instructions are then instrumented by a dynamic check that calls `reach_error()` before overflow arises. This reduces the problem to reachability checking solved above.

For `valid-free` and `valid-deref`, SYMBIOTIC employs static analysis to identify instructions that can potentially violate these properties and then uses

* Jury member and the corresponding author: martin.jonas@mail.muni.cz

slicing to remove the code that do not influence the execution of these instructions. Finally, it calls the symbolic executor JETKLEE that can verify these properties. For `valid-memtrack` and `valid-memcleanup`, we also rely on JETKLEE that tracks all allocated memory.

Symbolic execution with predicate abstraction The biggest conceptual change in SYMBIOTIC 11 is the addition of *symbolic execution with implicit predicate abstraction* (SEWPA) [11]. Instead of computing the abstract system that results from the predicate abstraction explicitly, the technique modifies symbolic execution to implicitly explore executions of the abstract system.

Given a set of *abstraction locations* that includes some location of each program loop and a *precision function* that assigns a set of predicates to each abstraction location, SEWPA performs standard symbolic execution with two modifications: (i) it performs predicate abstraction at each abstraction location and (ii) it stops exploring paths with a repeated abstract state. More precisely:

1. When an abstraction location l is encountered during symbolic execution, the entire symbolic memory is reset (replaced with fresh symbols). The path condition is then extended with constraints that force the new symbolic memory to agree with the original one on the values of all predicates at that location. This allows the execution to continue with any state that is equivalent in the abstract system.
2. After each such abstraction step, the path condition is extended with constraints encoding that the new symbolic memory evaluates at least one predicate differently than all the symbolic memories of all states in the same location l that occur on the current path.

This has a crucial benefit over standard SE: for any given set of predicates, the set of explored paths is *finite*. SEWPA can thus prove safety of programs with unbounded loops. When a counterexample is found, it needs to be checked for feasibility. We use the standard interpolation-based technique to compute new predicates from an unsatisfiable feasibility query. This blocks the spurious counterexample and refines the abstraction [9].

For detailed explanation of the technique, see the original paper [11] or the thesis [13] that describes our implementation of the technique into the SLOW-BEAST symbolic executor, which is used by SYMBIOTIC besides JETKLEE.

Non-termination checking SYMBIOTIC 11 improves non-termination proving and witness generation. Previously, SYMBIOTIC implemented a simplified version of the liveness-to-safety reduction [3] that stored values of program variables that can be modified by the loop before each loop iteration. If the values of these variables were the same after the iteration, the program was declared non-terminating. This only allowed finding non-terminating lassos that have a loop of one iteration. SYMBIOTIC 11 now implements the full liveness-to-safety reduction, with a non-deterministic choice whether the previous values of variables should be stored before each iteration of each loop. This allows finding also

lassos with longer loops, which is complete for finite-state systems. Additionally, we now generate non-termination witnesses in the new witness format [15].

2 Software Architecture

SYMBIOTIC consists of the main script that controls the verification process and of several reusable components responsible for processing the input program and for verification. On the front end, the input program is translated from C to LLVM IR by CLANG. All the subsequent components work on the LLVM IR level.

The internal components are (1) `llvm-instrumenter`, which performs instrumentation of LLVM IR, (2) library `dg` [5] computing dependency graphs for LLVM IR, and (3) `llvm-slicer`, which uses the dependency graphs to perform static slicing of LLVM IR. As external components, we rely on our fork of the symbolic executor KLEE [4], called JETKLEE, and our fork of the symbolic executor SLOWBEAST. We use JETKLEE for standard symbolic execution and SLOWBEAST for standard symbolic execution, BSELF, CSE, and SEWPA. Internally, both symbolic executors depend on SMT solver Z3. SLOWBEAST also uses MathSAT5 [8] for interpolant computation during refinement in SEWPA.

The main control script and SLOWBEAST are written in Python. All internal components and JETKLEE are written in C++ for efficiency.

3 Strengths and Weaknesses

Being based mainly on symbolic execution, SYMBIOTIC suffers from the well-known path explosion problem that makes the verification of programs with a large number of branchings and (unbounded) loops infeasible. This is partially mitigated by the static slicing, which removes some paths that cannot reach the error location and loops that do not affect the property. The problem is further mitigated by the implemented variants of symbolic execution, which reduce the number of explored paths and can deal with some long execution paths. However, SEWPA can only remove paths by pruning loop iterations; the technique cannot help with path explosion caused by control flow branchings alone. The implemented techniques are proven to be sound and the 7 incorrect answers are results of bugs and of imprecise models of some C functions.

To evaluate the changes in SYMBIOTIC 11, we compare the results of SYMBIOTIC 10.1 in SV-COMP 2025 and of SYMBIOTIC 11 in SV-COMP 2026. For fair comparison, we focus only on the benchmarks that were used in both of those years. For brevity, Table 1 shows only the results in the meta categories *C.ReachSafety* and *C.Termination*. It is clear that the changes significantly improve the number of solved benchmarks in most base categories. Thanks to SEWPA, SYMBIOTIC 11 correctly solves 203 more safe (verdict `true`) and 41 more unsafe (verdict `false`) benchmarks of *C.ReachSafety*.

The *Hardness* category is not listed as all its benchmarks have changed in SV-COMP 2026. Our evaluation [13] shows that SEWPA is particularly effective in this base category: it solves 628 benchmarks compared to 111 solved by SE.

Table 1. Numbers of solved benchmarks by SYMBIOTIC 10.1 and SYMBIOTIC 11 on the intersection of benchmarks from SV-COMP 2025 and SV-COMP 2026. The table is computed from the official results of SV-COMP 2025 and SV-COMP 2026.

Meta Category	Base Category	SYMBIOTIC 10.1	SYMBIOTIC 11
C.ReachSafety	Arrays	154	147
	BitVectors	31	38
	Combinations	222	220
	ControlFlow	26	29
	ECA	274	286
	Floats	362	381
	Hardware	88	99
	Heap	190	190
	Loops	534	594
	ProductLines	391	531
	Recursive	99	101
	Sequentialized	297	295
	XCSP	90	91
	Total	2758	3002
C.Termination	BitVectors	21	23
	MainControlFlow	63	77
	MainHeap	12	12
	Other	1152	1194
	Total	1248	1306

4 Tool Usage and Contributors

Binary form of SYMBIOTIC 11 is available on Zenodo [1], source code is available from <https://github.com/staticafi/symbiotic/releases/tag/svcomp26>. SYMBIOTIC can be executed as:

```
bin/symbiotic --sv-comp --prp <prpfile> [--32] <source>.
```

To compile SYMBIOTIC 11 from sources, we recommend using `Dockerfile` that is a part of the mentioned repository. For details, see the file `README.md`.

SYMBIOTIC 11 participates in all C categories of SV-COMP 2026, but fails for all benchmarks with the property `no-data-race` as it is not supported. The options above are used for all the categories.

SYMBIOTIC is developed at the AVELAB laboratory of Faculty of Informatics, Masaryk University. The tool is available under MIT license, including its internal components. External dependencies have permissive licenses, except for MathSAT5 [8], which is available for non-commercial use only.

Version 11 of SYMBIOTIC has been prepared by the authors of this system description under the supervision of Jan Strejček.

Acknowledgments. The authors were supported by the Czech Science Foundation project No. GA23-06506S.

Data Availability Statement. All data of SV-COMP 2026 are archived as described in the competition report [2] and available on the [competition web site](#). This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of SYMBIOTIC used in the competition is archived in the artifact [1] at Zenodo.

References

1. Ayaziová, P., Jonáš, M., Mihalkovič, V., Sedláček, J., Strejček, J.: Symbiotic: Submission to SV-COMP 2026 (Nov 2025). <https://doi.org/10.5281/zenodo.17698724>
2. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (Report on SV-COMP 2026). In: Proc. TACAS (2). Lecture Notes in Computer Science, vol. 16506. Springer (2026)
3. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. In: FMICS. Electronic Notes in Theoretical Computer Science, vol. 66, pp. 160–177. Elsevier (2002), [https://doi.org/10.1016/S1571-0661\(04\)80410-9](https://doi.org/10.1016/S1571-0661(04)80410-9)
4. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI 2008. pp. 209–224. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
5. Chalupa, M.: DG: analysis and slicing of LLVM bitcode. In: ATVA 2020. Lecture Notes in Computer Science, vol. 12302, pp. 557–563. Springer (2020), https://doi.org/10.1007/978-3-030-59152-6_33
6. Chalupa, M., Mihalkovič, V., Řechtáček, A., Zaoral, L., Strejček, J.: Symbiotic 9: String analysis and backward symbolic execution with loop folding (Competition Contribution). In: TACAS 2022. Lecture Notes in Computer Science, vol. 13244, pp. 462–467. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_32
7. Chalupa, M., Strejček, J.: Backward symbolic execution with loop folding. In: SAS 2021. Lecture Notes in Computer Science, vol. 12913, pp. 49–76. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_3
8. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: TACAS 2013. Lecture Notes in Computer Science, vol. 7795, pp. 93–107. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
9. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL 2004. pp. 232–244. ACM (2004)
10. Jonáš, M., Kumor, K., Novák, J., Sedláček, J., Trtík, M., Zaoral, L., Ayaziová, P., Strejček, J.: Symbiotic 10: Lazy memory initialization and compact symbolic execution (competition contribution). In: TACAS 2024. Lecture Notes in Computer Science, vol. 14572, pp. 406–411. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_29
11. Jonáš, M., Strejček, J., Griggio, A.: Combining symbolic execution with predicate abstraction and CEGAR. In: FMCAD 2024. pp. 271–280. IEEE (2024). https://doi.org/10.34727/2024/ISBN.978-3-85448-065-5_33
12. King, J.C.: Symbolic execution and program testing. Communications of ACM **19**(7), 385–394 (1976)
13. Sedláček, J.: Symbolic Execution with Predicate Abstraction in Slowbeast. Master’s thesis, Masaryk University, Faculty of Informatics, Brno, <https://is.muni.cz/th/zns01/>, supervisor: Martin Jonáš

14. Slaby, J., Strejček, J., Trtík, M.: Compact symbolic execution. In: ATVA 2013. Lecture Notes in Computer Science, vol. 8172, pp. 193–207. Springer (2013). https://doi.org/10.1007/978-3-319-02444-8_15, https://doi.org/10.1007/978-3-319-02444-8_15
15. Ádám, Z., Ayaziová, P., Bajczi, L., Beyer, D., Jankola, M., Lingsch-Rosenfeld, M., Strejček, J.: Non-termination witnesses and their validation. In: ASE 2025. IEEE (2025), <https://www.sosy-lab.org/research/non-termination-witnesses/>

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





ULTIMATE AUTOMIZER with a One-Dimensional Memory Model (Competition Contribution)

Manuel Bentele¹, Max Barth², Marcel Ebbinghaus¹, Jan Körner¹,
Daniel Dietsch¹, Matthias Heizmann^{3,*}, Dominik Klumpp⁴,
Frank Schüssele¹, and Andreas Podelski¹

¹ University of Freiburg, Freiburg, Germany

² LMU Munich, Munich, Germany

³ University of Stuttgart, Stuttgart, Germany

⁴ École Polytechnique, Paris, France

Abstract. Modeling memory accurately is crucial for the verification of C programs, but its inherent complexity limits scalability and efficiency. A memory model that abstracts unnecessary details while preserving essential information can significantly improve verification performance. In this work, we present a one-dimensional memory model that has recently been integrated into all software verifiers of the ULTIMATE tool family. Compared to our previous two-dimensional memory model, this abstraction trades some precision for improved efficiency. While it is not suitable for verifying memory safety properties, it enables more scalable reachability verification. An experimental evaluation on SV-COMP reachability benchmarks demonstrates that ULTIMATE AUTOMIZER solves up to 30.45 % more tasks when using the one-dimensional memory model.

Keywords: Memory Model · C Language · Formal Verification · Software Model Checking · ULTIMATE Program Analysis Framework.

1 Introduction

Memory modeling is a central challenge in the reachability verification of C programs. Low-level language features, such as pointer arithmetic, unions, and explicit dynamic memory management, complicate reasoning about a program's memory behavior and often limit the scalability of verification tools.

The tools of the ULTIMATE program analysis framework [16,17] have relied on a two-dimensional memory model inspired by symbolic block-based heap representations [15]. This model represents memory as a collection of blocks with explicit offsets, enabling precise reasoning about memory accesses and allocation. However, this expressiveness comes at the cost of substantial computational overhead, particularly for programs that intensively manipulate memory.

* Jury member for ULTIMATE AUTOMIZER: Matthias Heizmann.

In this work, we present a simplified one-dimensional memory model, inspired by [13], which has recently been integrated into the **ULTIMATE** tool family. By collapsing block and offset information into a single address space, the model reduces the complexity of memory reasoning. While this abstraction is not suitable for verifying memory safety and exhibits certain unsound behaviors, an experimental evaluation demonstrates that it significantly improves performance for reachability verification tasks. We did this evaluation with the software model checker **ULTIMATE AUTOMIZER** [8,12], which implements an automata-based verification approach [9]. Apart from minor bug fixes and extended support for C library functions, the new memory model was the only major change in **ULTIMATE AUTOMIZER** since the last **SV-COMP** [3].

2 **ULTIMATE**'s Two-dimensional Memory Model

The two-dimensional memory model, depicted in Figure 1, represents the memory of a C program as an unordered collection of distinct blocks, where each block consists of an ordered, finite list of cells. Each memory location is addressed by a pair (b, o) , where the base b identifies a block and the offset o denotes a specific cell within that block. Offsets start at zero and range over either integers or bitvectors. We represent the memory as a two-dimensional array, with the first dimension corresponding to blocks and the second to offsets within blocks.

While allocating new memory, we obtain new addresses non-deterministically as follows. To track allocated blocks, we maintain an auxiliary Boolean array **Valid**, indexed by address bases. A block is considered allocated if and only if its corresponding entry in **Valid** is true. When allocating new memory, the base of the new address must refer to an unallocated block, i.e., its entry in **Valid** is false prior to allocation. Upon allocation, this entry is updated to true. To determine whether a particular cell is allocated, we use a second auxiliary array **Length**, which maps each address base to a natural number representing the size of the corresponding block. The **Length** array is updated during allocation and defines the valid range of offsets for each allocated block. Upon deallocation, the corresponding entry in the **Valid** array is reset to false, thereby indicating that the block and all addresses with this base are once again available for allocation.

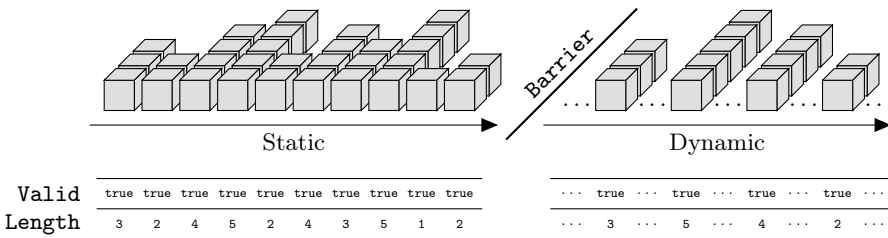


Fig. 1. Visualization of the memory space in the two-dimensional memory model.

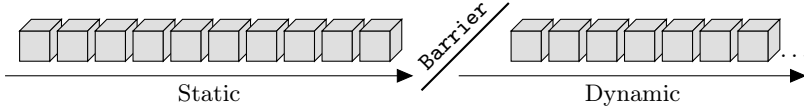


Fig. 2. Visualization of the memory space in the one-dimensional memory model.

As an optimization, we use integers or bitvectors as the datatype for address bases and partition the first dimension of memory into two disjoint regions, namely *Static* and *Dynamic*. The first region is reserved for the n statically allocated blocks that are allocated before entering the `main` function. The i -th statically allocated block is assigned base i , while the value 0 is reserved for the null pointer. To separate static from dynamic allocation, we introduce an auxiliary variable `Barrier`, which is assumed to be greater than n . Whenever dynamic memory allocation non-deterministically returns an address, its base is required to be strictly greater than `Barrier`, ensuring that dynamically allocated blocks do not overlap with statically allocated ones.

3 ULTIMATE’s One-dimensional Memory Model

In the one-dimensional memory model, depicted in Figure 2, memory addresses are represented as a single `int` or bitvector rather than as base–offset pairs. To ensure that each allocation yields fresh, non-overlapping addresses, we maintain an auxiliary variable that tracks the lowest address that has not been allocated so far. This variable is initialized to 1 and is incremented by the size of the allocated block after each allocation. Similar to the two-dimensional model, an optimization partitions memory into *Static* and *Dynamic* regions, where the addresses in the *Static* region are fixed at compile time. Unlike the previous model, there are no auxiliary arrays `Valid` and `Length` for tracking allocated memory. As a result, allocated addresses are never reclaimed, and deallocation is modeled as a no-op, i.e., it has no effect on the memory state.

4 Strengths and Weaknesses

We evaluate the implementations of the one- and two-dimensional memory models in ULTIMATE AUTOMIZER [8,12] using benchmarks from the benchmark set of SV-COMP 2026 [4]. Our evaluation comprises 18 709 benchmarks from the meta-categories *Concurrency*, *ReachSafety* and *SoftwareSystems*, for which the property *unreach-call* is checked. Overall, ULTIMATE AUTOMIZER solves 8.05% more benchmarks when employing the one-dimensional memory model, which is an increase of 387 solved tasks. This improvement includes both, safe benchmarks (from 3475 to 3795) and unsafe benchmarks (from 1333 to 1400).

A strength of the one-dimensional memory model is its significant performance improvement in verification tasks involving many memory operations.

This advantage is most evident in the *DeviceDriversLinux64* sub-category within the *SoftwareSystems* meta-category, where programs heavily interact with memory. As a result, 30.45 % more tasks are solved. In tasks with few or no memory operations, performance is comparable to that of the two-dimensional model, indicating no significant difference. Hence, we implement the one-dimensional memory model in ULTIMATE AUTOMIZER for SV-COMP 2026 to verify not only the *unreach-call* property but all other properties except memory safety ones.

A limitation of the one-dimensional memory model is that it is not suitable for verifying memory safety properties, as it abstracts away essential aspects of C memory management, including precise allocation, deallocation, and bounds checking. Furthermore, the model exhibits two sources of unsoundness. First, when addresses are modeled using unbounded integers, each memory cell can be allocated at most once. As a result, erroneous executions in which a deallocated memory cell is later reallocated at the same address are not captured. Second, when addresses are modeled using bitvectors, allocation may wrap around after reaching the maximal representable value. As a consequence, the same address may be allocated multiple times, introducing spurious aliasing between distinct dynamic allocations. Our evaluation unveils a total of 10 incorrect results, where we suspect the issues are due to invalid verification tasks rather than limitations of the memory model. Further investigation is needed to confirm this.

5 Architecture, Setup, Configuration, and Project

The software model checker ULTIMATE AUTOMIZER [8,12] is part of the open-source program analysis framework ULTIMATE [16,17], which is implemented in Java and licensed under LGPLv3.⁵ The ULTIMATE framework has received contributions from over 60 developers across multiple universities and institutions. The submitted release version 0.3.1-dev-00d43373f5 requires Java 21 and Python 3.6. The Linux package for this release includes precompiled binaries of the required SMT solvers Z3 [10,11], CVC4 [1,14], and MathSAT [5,7]. The Linux package is available as a ZIP archive on Zenodo [6] and also contains the Python script `Ultimate.py` to invoke ULTIMATE AUTOMIZER via the script call

```
./Ultimate.py -spec <p> -file <f> -architecture <a> -full-output
```

where `<p>` is an SV-COMP property file, `<f>` is an input C file, `<a>` is the architecture (32bit or 64bit), and `-full-output` enables verbose output to `stdout`. A witness is saved in the files `witness.graphml` and `witness.yml`. The benchmarking tool BENCHEXEC [2] supports ULTIMATE AUTOMIZER via its tool-info module `ultimateautomizer.py`.⁶ ULTIMATE AUTOMIZER participates in all categories, as declared in its benchmark definition file `uautomizer.xml`.⁷

⁵ GNU Lesser General Public License, version 3: <https://www.gnu.org/licenses/lgpl-3.0.html> (retrieved: 06/12/2025)

⁶ BENCHEXEC tool-info module: <https://github.com/sosy-lab/benchexec/blob/main/benchexec/tools/ultimateautomizer.py> (retrieved: 06/12/2025)

⁷ BENCHEXEC benchmark definition: <https://gitlab.com/sosy-lab/sv-comp/benchmark-defs/-/blob/main/benchmark-defs/uautomizer.xml> (retrieved: 06/12/2025)

Data Availability Statement. The competition contribution for ULTIMATE AUTOMIZER is available as a ZIP archive on Zenodo [6].

References

1. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proc. CAV. pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
2. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. STTT **21**(1), 1–29 (Nov 2017). <https://doi.org/10.1007/s10009-017-0469-y>
3. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS. pp. 151–186. Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_9
4. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In: Proc. TACAS. Springer (2026)
5. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT 5 SMT solver. In: Proc. TACAS. pp. 93–107. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
6. Dietsch, D., Bentele, M., Ebbinghaus, M., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: ULTIMATE AUTOMIZER SV-COMP 2026 competition contribution (Nov 2025). <https://doi.org/10.5281/zenodo.17735224>
7. Fondazione Bruno Kessler, University of Trento: MathSAT 5 website (2025), <https://mathsat.fbk.eu>, retrieved: 06/12/2025
8. Heizmann, M., Chen, Y.F., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: ULTIMATE AUTOMIZER and the search for perfect interpolants (competition contribution). In: Proc. TACAS. pp. 447–451. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30
9. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
10. Microsoft Research: Z3 website (2025), <https://www.microsoft.com/en-us/research/project/z3>, retrieved: 06/12/2025
11. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS. pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
12. Schüssele, F., Bentele, M., Dietsch, D., Heizmann, M., Jiang, X., Klumpp, D., Podelski, A.: ULTIMATE AUTOMIZER and the abstraction of bitwise operations (competition contribution). In: Proc. TACAS. pp. 418–423. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_31
13. Sinz, C., Falke, S., Merz, F.: A precise memory model for low-level bounded model checking. In: Proc. SSV. USENIX Association (2010), <https://www.usenix.org/conference/ssv10/precise-memory-model-low-level-bounded-model-checking>
14. Stanford University, University of Iowa: CVC4 source code (2021), <https://github.com/CVC4/CVC4-archived>, retrieved: 06/12/2025
15. Trtík, M., Strejček, J.: Symbolic memory with pointers. In: Proc. ATVA. pp. 380–395. Springer (2014). https://doi.org/10.1007/978-3-319-11936-6_27
16. University of Freiburg: ULTIMATE source code (2025), <https://github.com/ultimate-pa/ultimate>, retrieved: 06/12/2025
17. University of Freiburg: ULTIMATE website (2025), <https://ultimate-pa.org>, retrieved: 06/12/2025

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





ULTIMATE PARALIZER: Parallel Trace Abstraction (Competition Contribution)*

Max Barth**¹ , Daniel Dietsch² , Matthias Heizmann³ , and
Marie-Christine Jakobs¹

¹ LMU Munich, Munich, Germany

² University of Freiburg, Freiburg, Germany

³ University of Stuttgart, Stuttgart, Germany

Abstract. We present the multi-threaded software model checker `ULTIMATE PARALIZER`, which lets us use multiple CPU cores effectively. `ULTIMATE PARALIZER` is built on `ULTIMATE AUTOMIZER`'s automata-theoretic approach to software model checking, which performs a variant of counterexample-guided abstraction refinement. The parallelization idea of `ULTIMATE PARALIZER` is simple but effective. For parallelization, `ULTIMATE PARALIZER` analyzes multiple, diverse counterexample independently in parallel including the computation of an abstraction refinement if necessary. A central coordinator steers the verification process, in particular it selects the counterexamples for analysis and performs the necessary abstraction updates. In SV-COMP 2026, `ULTIMATE PARALIZER` participates in the categories reach-safety and no-overflows and in the demo wall-time track.

Keywords: `ULTIMATE AUTOMIZER` · Parallel Trace Abstraction · Software verification · Software model checking · Automata

1 Software Verification with Parallel Trace Abstraction

While software verification is a powerful method to check programs against (safety) properties, it may be time-consuming. One solution to reduce the wall time of a verifier is to leverage modern multi-core CPUs.

We present the tool `ULTIMATE PARALIZER` that realizes a parallel version of `ULTIMATE AUTOMIZER`'s [3] automata-theoretic approach to software model checking [5,6]. To this end, it implements parallel trace abstraction [2]. Like the sequential implementation of automata-based software model checking [5,6], parallel trace abstraction maintains a so-called trace abstraction, an automaton that accepts all not yet analyzed, syntactic error trace (i.e., counterexamples) of the program. To analyze program safety, our tool checks the feasibility of error traces, reports UNSAFE, i.e., property violated, if an error trace is proven semantically feasible, and refines the trace abstraction for spurious error traces,

* This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 378803395 (ConVeY) and 496852682 (ReVeriX).

** Jury Member: Max Barth

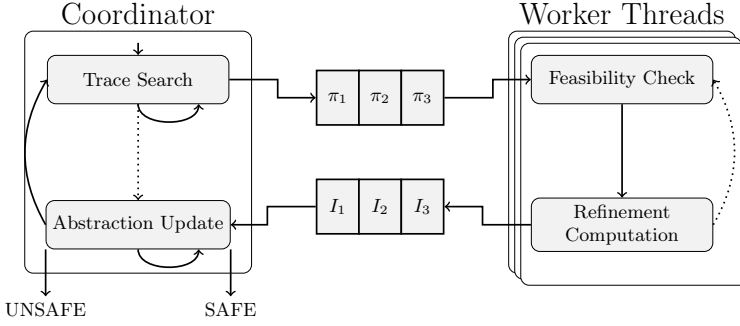


Fig. 1. Overview of ULTIMATE PARALIZER's workflow

i.e., it removes infeasible error traces including the analyzed, spurious error trace. For refinement, the tool computes an interpolant automaton [5] that accepts the spurious error trace and error traces that are infeasible due to similar reasons. The tool stops with verification result SAFE, i.e., property satisfied, if the abstraction becomes empty, i.e., all error traces in the program have been proven semantically infeasible. Instead of analyzing the error traces, i.e., counterexamples, one after the other, parallel trace abstraction may analyze different error traces in parallel. To this end, a central coordinator steers the verification procedure. The coordinator searches for error traces and forwards them for analysis to an available worker thread, updates (i.e., refines) the trace abstraction based on refinement suggestions (i.e., interpolant automata) computed by the workers, and determines the verification result. Workers analyze error traces, i.e., they check their feasibility, compute refinement suggestions for infeasible error traces in form of interpolant automata, and report their results to the coordinator.

Figure 1 shows the workflow of ULTIMATE PARALIZER sketched above. We notice the single coordinator on the left and the multiple workers on the right. Coordinator and worker exchange information via buffers. The coordinator provides error traces π_i for analysis and the workers provide their analysis results and for spurious error traces, also interpolant automata I_i . Both, workers and the coordinator, perform iterative processes and block if their input buffer is empty (indicated by the dotted edges). We explain their detailed workflow in the following paragraphs.

Workflow of the Coordinator. The coordinator iteratively executes two subsequent phases until it is able to determine the verification result. In its first phase, it iteratively searches for distinct error traces in the abstraction (see below for details). The first phase stops if the coordinator found a distinct error trace for each idle worker or no further error traces exist in the abstraction. The second phase processes worker results. It starts as soon as a worker result becomes available and ends at latest when the buffer with the worker results becomes empty. In each iteration of the second phase, the coordinator inspects one worker result. If the result reports a feasible error trace, the coordinator

stops the verification with result UNSAFE. Otherwise, the coordinator updates its abstraction with the provided interpolant automaton, formally it computes the difference of its current trace abstraction and the interpolant automaton. If the language of the trace abstraction becomes empty after the update, the coordinator stops the verification with result SAFE.

Workflow of a Worker. Workers analyze error traces. In each of its iteration steps, the worker first takes an error trace provided by the coordinator and then checks its semantic feasibility with the help of an SMT solver. If the worker determines that the error trace is feasible, it provides this result to the coordinator and then finishes its current iteration. If the error trace is infeasible, the worker uses interpolation to compute a refinement suggestion, i.e., an interpolant automaton [5], which accepts the infeasible error trace and error traces with similar reasons for infeasibility. The interpolant automaton guarantees that all error traces it accepts are semantically infeasible and, thus, may be safely excluded from the not yet analyzed error traces, i.e., the trace abstraction maintained by the coordinator. After computing the interpolant automaton, the worker provides it together with the infeasibility result to the coordinator and continues with its next iteration.

Diverse Trace Search Now, we discuss the trace search of the coordinator, which needs to ensure that we provide distinct error traces to different workers. To this end, the coordinator remembers the traces that are currently analyzed by the workers, i.e., the traces it added to its output buffer but has not received a result for. During its search for an error trace in its trace abstraction, it uses its remembered traces to steer the search. More concretely, it uses a combination of depth-first search and breadth-first search. First, it uses a depth-first search that aims to construct a prefix of an error trace that has not yet been analyzed. When exploring the successors of a path, the depth-first search considers all remembered error traces following that path and explores successors that are taken less often by these error traces first. Once a not yet seen prefix has been found, the search tries to complete it to an error trace using breadth-first search. If an acceptable error trace is found, it will be returned otherwise the search backtracks. For more details on the search algorithm, we refer to the publication on parallel trace abstraction [2]. In practice, we limit the time spent in the search algorithm if error traces are still processed by workers. This allows us to prioritize result processing over trace search.

2 Discussion of Strengths and Weaknesses

We have observed that our parallelization may significantly reduce the wall time, when limiting the CPU time or when limiting the wall time. For a few tasks (less than 100), e.g., `square_4+soft_float_1-3a.c.cil.c`, we observed speed-ups beyond Amdahl’s law, i.e., parallelization saved actual work. Nevertheless, reductions are likelier for long running verification tasks. In addition, trace abstraction needs to perform multiple refinements, i.e., requires the analysis of multiple error traces, to effectively make use of parallelization. Otherwise, work-

ers may perform redundant work. In practice, trace abstraction solves many verification tasks at SV-COMP with less than six refinements. For those, there is little opportunity to really profit significantly from parallel trace abstraction. The diverse trace search simply cannot provide us with enough error traces that allow us to prove distinct parts of the program. Nevertheless, even if we do not achieve a reduction in wall time, the additional time required by parallel trace abstraction is typically in the range of a few seconds. In general, parallel trace abstraction comes at the cost of increased memory usage. In the past, the median increase for six worker threads was lower than double the memory for sequential verification and we do not get more out of memory results in SV-COMP than sequential verification.

Since parallel trace abstraction shares the available CPU time among threads, we observed a reduction in the tasks solvable with increasing number of worker threads. The reason is that the CPU time an individual worker may spend decreases. Hence, we may run out of CPU time before a worker finishes the analysis of a time-consuming error trace. If we use a wall-time limit, we will not have these effects. Indeed, we typically solve more tasks with increasing number of worker threads while using a wall-time limit.

3 Setup and Configuration

ULTIMATE PARALIZER is part of the Ultimate framework⁴, which is licensed under LGPLv3. In SV-COMP 2026, ULTIMATE PARALIZER participates in the categories ReachSafety and NoOverflows as well as the wall-time demo track.

To run ULTIMATE PARALIZER in its SV-COMP 2026 version [1], one requires Java 21 and Python 3.6 and needs to execute the following command:

```
./Ultimate.py --spec <p> --file <f> --architecture <a> --full-output .
```

Thereby, parameter <p> refers to a SV-COMP property file, which determines the checked property. Parameter <f> specifies the input C file to be verified, and parameter <a> determines the architecture (32bit or 64bit). Furthermore, option --full-output ensures that ULTIMATE PARALIZER generates verification witnesses. To choose the number <n> of worker threads that ULTIMATE PARALIZER should use, one must extend the above command line with the option

```
--traceabstraction.threadlimit.for.parallel.cegar <n> .
```

In SV-COMP 2026 [4], we use two worker threads for the categories ReachSafety and NoOverflows, which are restricted by CPU time limit, and seven worker threads in the wall-time demo track. Thus, we ensure that we have at least one CPU core to let Java perform garbage collection.

⁴ <https://ultimate.informatik.uni-freiburg.de> and <https://github.com/ultimate-pa/ultimate>

Data Availability The SV-COMP 2026 version of ULTIMATE PARALIZER is available on Zenodo [1]. Its corresponding FM-tools entry is available on GitLab⁵.

References

1. Barth, M.: Ultimate Paralyzer at SV-Comp 2026. Zenodo (2025), <https://doi.org/10.5281/zenodo.17688158>
2. Barth, M., Jakobs, M.C.: Multi-threaded software model checking via parallel trace abstraction refinement (2026), <https://arxiv.org/abs/2509.13699v2>
3. Bentele, M., Barth, M., Ebbinghaus, M., Körner, J., Dietsch, D., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: ULTIMATE AUTOMIZER with a one-dimensional memory model (competition contribution). In: Proc. TACAS (2), pp. xx–yy. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_37
4. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026). In: Proc. TACAS (2), pp. xx–yy. LNCS 16506, Springer (2026). https://doi.org/https://doi.org/10.1007/978-3-032-22749-2_23
5. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Proc. SAS. pp. 69–85. LNCS 5673, Springer (2009). https://doi.org/10.1007/978-3-642-03237-0_7
6. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



⁵ <https://gitlab.com/sosy-lab/benchmarking/fm-tools/-/blob/main/data/uparalizer.yml>

Author Index

A

Amat, Nicolas 85
Amir, Guy 85
Amrollahi, Daneshvar 109
Arceri, Vincenzo 543
Ayaziová, Paulína 523, 583
Ayoun, Sacha-Élie 195

B

Bae, Kyungmin 23
Bajczy, Levente 510
Barbone, Mark 85
Barrett, Clark 109, 151
Barth, Max 589, 595
Bentele, Manuel 589
Bergström, Axel 3
Beyer, Dirk 42, 461
Bianchi, Filippo 543
Blicha, Martin 530
Brablec, Tomáš 571

C

Chareton, Christophe 419
Chen, Bo-Hung 440
Chen, Chen 309
Chen, Tian-Fu 440
Chen, Yu-Fang 419
Chien, Po-Chun 42
Chiou, Dah-Wei 440
Chung, Kai-Min 419
Cleaveland, Rachel 151
Coughlin, Nicholas 235

D

Dacík, Tomáš 571
Darulova, Eva 3
Deng, Chenhui 309
Di Biase, Paolo 536
Dietsch, Daniel 589, 595
Dill, David 109
Dobos-Kovács, Mihály 510
Dyer, Robert 277

E

Ebbinghaus, Marcel 589
Eisenbarth, Thomas 503, 577
Elad, Neta 353
Emoto, Kento 215
Erhard, Julian 516

F

Ferrara, Pietro 543
Fischer, Bernd 536
Frohme, Markus 394
Frohn, Florian 131

G

Gardner, Philippa 195
Giesl, Jürgen 131
Giesl, Peter 131
Gilot, Andrea 3
Glitta, Oliver 530

H

Hamana, Makoto 215
Heizmann, Matthias 589, 595
Herbretau, Frédéric 373
Holter, Karoliine 516, 523
Hsieh, Min-Hsiu 419
Hu, Guangyu 334
Huang, Bo-Yuan 42
Huang, Wei-Jia 419

I

Inverardi, Paola 65

J

Jacobs, Jules 85
Jakobs, Marie-Christine 595
Jiang, Jie-Hong R. 440
Jonáš, Martin 559, 583

K

Karmios, Nat 195
Kaufmann, Daniela 309

Klumpp, Dominik 589
 Kocal, Ali Rasim 173, 516
 Kofroň, Jan 530
 Körner, Jan 589
 Kovács, Dániel 510

L

La Torre, Salvatore 536
 Laarman, Alfons 419
 Lahiri, Shuvendu 109
 Lee, Nian-Ze 42
 Lemberger, Thomas 42
 Lisovenko, Teodors 543
 Lommen, Nils 131
 Loose, Nils 503, 577
 Lotan, Raz 353
 Lu, Shan 109

M

Ma, Chien-Kai 440
 Mächtle, Felix 503, 577
 Marin, Mircea 23
 Matheja, Christoph 289
 Mei, Jingyi 419
 Michael, Alicia 235
 Mihalkovič, Vincent 583
 Milanese, Marco 551
 Miné, Antoine 551
 Mirani, Mahrokh 65
 Molnár, Vince 510
 Moloney, Charles 277
 Monat, Raphaël 551
 Mondok, Milán 510
 Moussaoui Remil, Naïm 565

N

Negrini, Luca 543
 Nicol, John 394

O

Olivieri, Luca 543
 Ölveczky, Peter Csaba 23
 Ouadjaout, Abdelraouf 551

P

Padon, Oded 353
 Panneke, Lukas 257
 Parlato, Gennaro 536
 Pelliccione, Patrizio 65

Petter, Michael 516
 Pettinau, Roberto 289
 Podelski, Andreas 589
 Point, Gérald 373

R

Raimondi, Franco 65
 Reja, Mario 23
 Rukhaia, Mikheil 23

S

Saan, Simmo 173, 516, 523
 Schrammel, Peter 536
 Schüssele, Frank 589
 Schwarz, Michael 173, 516
 Sedláček, Jindřich 583
 Seidl, Helmut 173, 516
 Serr, Jan-Niclas 503
 Sherman, Elena 277
 Shoham, Sharon 353
 Sieck, Florian 577
 Štěpková, Adéla 559
 Song, Zhan 309
 Strejček, Jan 461, 523, 559, 583
 Sun, Chuyue 109
 Sun, Yican 109

T

Telbisz, Csanád 510
 Tobler, James 235
 Tork, Sadra Bayat 235
 Troquard, Nicolas 65

U

Urban, Caterina 565

V

Viswanathan, Gautham 373
 Vojdani, Vesal 516, 523
 Vojnar, Tomáš 571

W

Walukiewicz, Igor 373
 Wehrheim, Heike 257
 Winter, Kirsten 235

Y

Yu, Cunxi 309

Z

Zanatta, Giacomo [543](#)

Zhang, Ethan [109](#)

Zhang, Hongce [309](#), [334](#)

Zhang, Wei [334](#)

Zhou, Xiaofeng [334](#)